UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

GRADO EN INGENIERÍA INFORMÁTICA

ESPECIALIDAD EN INGENIERÍA DEL SOFTWARE

MEMORIA

Integración de software contenerizado en COMPSs

Autor:
Carlos Santiago Alvarado Aguilera

Directora del Proyecto:
Rosa Maria Badia Sala
Codirector del Proyecto:
Jorge Ejarque Artigas

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Índice

Lista de figuras

Índice de tablas

1

Contexto

Este proyecto es un Trabajo Final de Grado(TFG) de la especialidad de Ingeniería del Software, de la Facultad de Informática de Barcelona (Universidad Politécnica de Cataluña). Se trata de un proyecto de modalidad A en colaboración con el grupo Workflows and Distributed Computing [1] del Centro Nacional de Supercomputación (BSC) [2]; en el que se ha realizado el desarrollo e implementación de un prototipo, que permita al *Framework COMP Superscalar* [3] gestionar la combinación de software contenerizado para integrar diferentes piezas de software en una sola aplicación.

1.1. Introdución

El tamaño y la complejidad de las infraestructuras paralelas y distribuidas actuales plantean un importante desafío a la programación. Algunas de estas dificultades son inherentes a la programación concurrente y distribuida, por ejemplo, tratar con varios threads, gestionar recursos y sincronizar datos. Por otro lado tenemos la portabilidad del software, que pone en riesgo el funcionamiento de nuestras aplicaciones en algunas plataformas de proveedores cloud del mercado. Por esto, existe una gran necesidad de modelos de programación y lenguajes que faciliten desarrollar aplicaciones paralelas y distribuidas, con el fin de aumentar la productividad de los programadores en las infraestructuras actuales, sin sacrificar su rendimiento.

COMP Superscalar, a partir de ahora COMPSs [3] proporciona un modelo de programación y un *Runtime*[1] que tiene como objetivo resolver los problemas mencionados anteriormente, facilitando el desarrollo de aplicaciones y su ejecución en entornos distribuidos. COMPSs implementa un modelo de programación basado en tareas, que permite que las aplicaciones se escriban siguiendo un paradigma secuencial, donde el programador es responsable de identificar y definir las funciones que potencialmente se podrán ejecutar en paralelo. El *Runtime* de COMPSs se encarga de detectar todas las tareas y las dependencias de datos entre ellas, para luego lanzarlas de manera concurrente en los recursos disponibles. Estos recursos pueden ser nodos en *clusters* y *grids*, o máquinas virtuales o contenedores en *clouds*.

La otra característica importante del *framework* de COMPSs es la capacidad de ejecutar las aplicaciones de forma transparente con respecto a la infraestructura subyacente. Un aspecto clave de proporcionar un modelo de programación transparente a la infraestructura, es que los programas se desarrollan una vez y se ejecutan en múltiples servidores, sin tener que cambiar la implementación. Esto es importante cuando

[1] Sistema de gestión que se encarga de organizar varios aspectos de la ejecución de la aplicación.

se debe lograr la portabilidad entre clouds, ya que el programador no tendrá que lidiar con los detalles de un proveedor específico ni arquitecturas diferentes.

Otra herramienta que también aumenta la productividad de los programadores en entornos distribuidos, son los contenedores. Estos permiten empaquetar aplicaciones junto a sus dependencias, con el fin de proporcionar entornos aislados donde se puedan ejecutar diferentes piezas de software. En vez de virtualizar la pila de hardware como hacen las máquinas virtuales, los contenedores se virtualizan a nivel del sistema operativo y pueden ejecutarse más de uno encima del kernel al mismo tiempo. Esto significa que los contenedores son mucho más ligeros, se inician mucho más rápido y gestionan los recursos de una manera más eficiente. Gracias a estas particularidades podemos aislar las aplicaciones entre sí, a menos que los conectemos de forma explícita, evitando los conflictos de dependencias o disputa de recursos.

Gracias a la contenerización, los desarrolladores pueden crear entornos predecibles donde ejecutar sus aplicaciones. Estas pueden incluir dependencias de software, como versiones específicas de *frameworks*, lenguajes de programación o diferentes bibliotecas. De esta manera, se garantiza la portabilidad del software desarrollado en diferentes plataformas aportando productividad al programador. Por tanto, podemos decir que COMPSs y la contenerización son una combinación perfecta para mejorar la productivad de los programadores en entornos distribuidos.

Este TFG tiene como objetivo general el desarrollo e integración de un prototipo, que permita al *framework* COMPSs gestionar la combinación de software contenerizado, para permitir la integración de diferentes piezas de software en una sola aplicación.

Para la integración hemos elegido las plataformas *DOCKER* y *SINGULARITY*. Por una parte DOCKER, ya que es el más utilizado a día de hoy, provee las herramientas necesarias para crear, mantener, distribuir y desplegar imágenes de contenedores. Es fácil de instalar, existe mucha documentación disponible y una extensa comunidad de desarrolladores que utilizan esta plataforma. Además, COMPSs ya tiene una integración con esta plataforma que admite la ejecución de una aplicación distribuida en un *clúster Docker Swarm* [4][2], donde nuestro *framework* trabaja como una dependencia dentro de los contenedores. Por otro lado, hemos seleccionado *SINGULARITY*, porque al igual que DOCKER es portable y se puede distribuir. Es el más utilizado dentro de los *clusters* de supercomputación, en entornos donde no se pueden ejecutar procesos sin privilegios de administración y te permite acceder a drivers específicos del supercomputador. Además de soportar hardware y aplicaciones *HPC*[3] y una infinidad de tarjetas gráficas.

[2] *Docker Swarm* es una herramienta integrada en el ecosistema de Docker que permite orquestar un *cluster* de servidores.
[3] *High-Performance Computing*

1.1.1. Stakeholders

En este apartado se definirán los principales actores que participan en el proyecto, a quien va dirigido el producto y quien lo utilizará, es decir, todas las personas que tengan algún tipo de relación con él.

Directora y Codirector del proyecto

La directora del proyecto es Rosa Maria Badia Sala y el codirector es Jorge Ejarque Artigas. Estas personas se van a encargar de apoyar al desarrollador durante el proceso de diseño y desarrollo, guiando, validando y supervisando las tareas para cumplir con los objetivos.

Grupo Workflows and Distributed Computing

El Grupo Workflows and Distributed Computing del departamento de Computer Sciences del BSC, se verá directamente beneficiado, ya que sus actividades giran en torno al proyecto COMPSs, que es sobre el cual realizamos el desarrollo y aportamos valor.

Usuarios de COMPSs

Otra de las partes beneficiadas serían las empresas y desarrolladores que hacen uso del framework COMPSs en sus proyectos y aplicaciones.

Desarrollador del proyecto

En este caso, es el autor de este trabajo (TFG) y es el que se encarga de diseñar e implementar la nueva funcionalidad. Tendrá la ayuda del director que le guiará en todo lo que necesite. Además se encarga de hacer las pruebas necesarias para asegurarse de que el software funcione perfectamente.

1.2. Estado del arte

Como trabajaremos en la integración de una nueva funcionalidad dentro de COMPSs, se pasará a explicar su funcionamiento. Luego, considerando que gran parte de la solución es integrar software contenerizado con *DOCKER* y *SINGULARITY*, pasaremos a explicar qué es la contenerización.

1.2.1. Visión general de COMPSs

COMPSs [3] es un modelo de programación basado en tareas que pertenece a la familia de Frameworks con flujos de trabajo implícitos y tiene como objetivo facilitar la paralelización de aplicaciones. Soporta aplicaciones en Java y proporciona bindings para C/C++ y Python (PyCOMPSs [5]). Las aplicaciones se componen de un programa principal, que es un código secuencial y de funciones, donde es el usuario el principal responsable de identificar y definir las funciones que se ejecutarán como tareas paralelas. Estas tareas serán candidatas a ser ejecutadas en paralelo asincrónicamente entre ellas explotando el paralelismo inherente dentro de la aplicación. El runtime se encarga de detectar todas las tareas y las dependencias de datos entre ellas para luego lanzar estas tareas en los recursos disponibles. COMPSs también tiene mecanismos de tolerancia a fallas (como reenvío de jobs y reprogramación de tareas o recursos), tiene una herramienta de monitoreo a través de una interfaz web incorporada, admite instrumentación usando la herramienta Extrae [6] para generar trazas que pueden analizarse con Paraver [7], tiene un IDE de Eclipse [8] y conectores cloud.

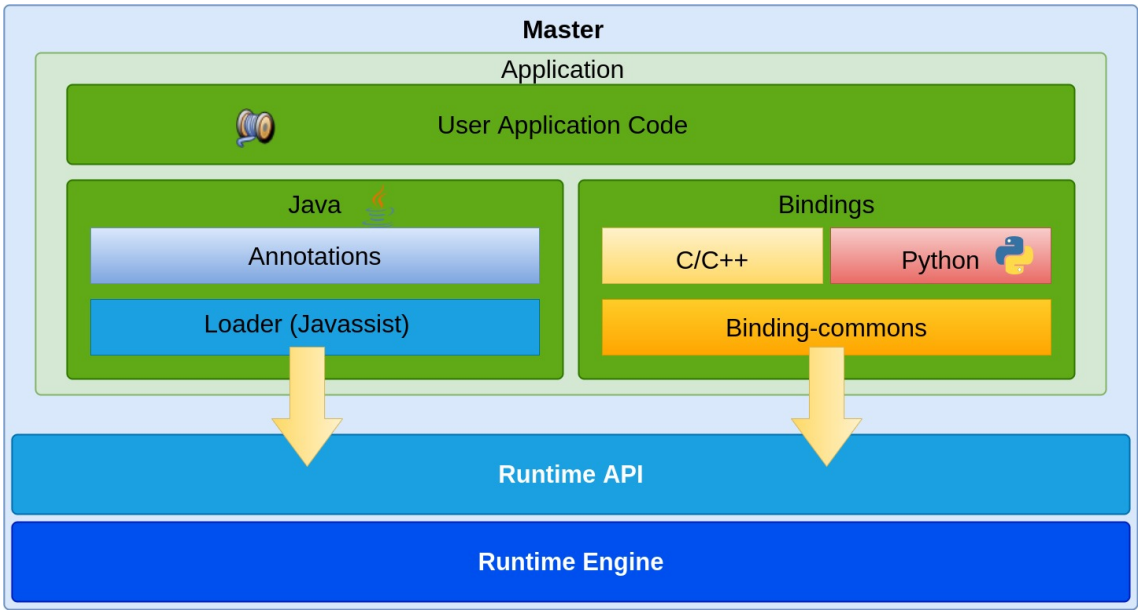


Figura 1.1: Visión general de COMPSs

El modelo de programación de COMPSs tiene tres características principales:

- **Programación secuencial:** Los usuarios no necesitan lidiar con ningún problema de paralelización ni gestión de recursos, como la creación de hilos, sincronización, distribución de datos, mensajes o tolerancia a fallas. Los programadores de COMPSs solo seleccionan los métodos que deben considerarse como tareas y el runtime los genera asíncronamente hacia los recursos que tenga disponible en lugar de ejecutarlos en local y secuencialmente.

- **Agnosticismo de la infraestructura:** El modelo COMPSs abstrae la aplicación de la infraestructura subyacente. Por lo tanto, las aplicaciones COMPSs no incluyen ningún detalle relacionado con la plataforma, como la implementación o la gestión de recursos. Esta característica hace que las aplicaciones sean portables entre diferentes infraestructuras.

- ***API***[4] **limitada:** Cuando se utiliza el lenguaje nativo de COMPSs, el lenguaje Java, el modelo no requiere ninguna *API* para la construcción y ejecución de tareas, ya que las tareas se definen mediante anotaciones de código. Considerando que COMPSs instrumenta el código de la aplicación en tiempo de ejecución para detectar las tareas, podemos desarrollar aplicaciones enteras con el lenguaje java y sus bibliotecas estándar. Con respecto a los enlaces (bindings) de Python y C/C++, se debe usar un pequeño conjunto de llamadas *API* para la sincronización de los generados por las tareas y el programa principal.

Modelo de programación

Para ilustrar cómo es el funcionamiento de las aplicaciones COMPS a partir de un código secuencial, presentamos la aplicación *incremento secuencial* en Java que aumenta *N* veces tres contadores diferentes. Cada incremento se desarrolla mediante una tarea separada. El propósito de esta aplicación es mostrar el paralelismo entre los tres contadores. A continuación, en la Figura 1.2, proporcionamos el código principal de esta aplicación.

```java
package increment;
public class Increment {

    public static void main(String[] args) throws Exception {
        // Check and get parameters
        if (args.length != 4) {
            usage();
            throw new Exception("[ERROR] Incorrect number of parameters");
        }
        int N = Integer.parseInt(args[0]);
        int counter1 = Integer.parseInt(args[1]);
        int counter2 = Integer.parseInt(args[2]);
        int counter3 = Integer.parseInt(args[3]);

        // Initialize counter files
        System.out.println("Initial counter values:");
        initializeCounters(counter1, counter2, counter3);

        // Print initial counters state
        printCounterValues();

        // Execute increment tasks
        for (int i = 0; i < N; ++i) {
            IncrementImpl.increment(fileName1);
            IncrementImpl.increment(fileName2);
            IncrementImpl.increment(fileName3);
        }

        // Print final counters state (sync)
        System.out.println("Final counter values:");
        printCounterValues();
    }
}
```

Figura 1.2: Increment main class

[4] Aplication Program Interface

Como se muestra en el código principal, esta aplicación tiene 4 parámetros:

1. ***N*:** Número de veces para incrementar el contador.
2. **InitialValue1:** Valor inicial para el contador 1.
3. **InitialValue2:** Valor inicial para el contador 2.
4. **InitialValue3:** Valor inicial para el contador 3.

Después de implementar el código secuencial, los usuarios deben crear una interfaz para transformar su aplicación secuencial en una aplicación COMPSs. El runtime requiere de esta interfaz para saber qué métodos debe ejecutar de forma remota como tareas. Las interfaces COMPSs deben definirse dentro de un archivo con el mismo nombre que la clase principal de la aplicación pero con el sufijo «*Itf*» (por ejemplo, en el ejemplo anterior, la interfaz debe definirse en el archivo *IncrementItf.java*). En cuanto a su contenido, la interfaz debe contener una entrada por tarea, que se decora con la anotación ***@Method***. En la Figura 1.3 proporcionamos el código de la interfaz «*Itf*».

```
package increment;
import integratedtoolkit.types.annotations.Parameter;
import integratedtoolkit.types.annotations.parameter.Direction;
import integratedtoolkit.types.annotations.parameter.Type;
import integratedtoolkit.types.annotations.task.Method;

public interface IncrementItf {

    @Constraints(computingUnits = "1")
    @Method(declaringClass = "increment.IncrementImpl")
    void increment(
        @Parameter(type = Type.FILE, direction = Direction.INOUT) String file
    );
}
```

Figura 1.3: Increment Interface

Para cada anotación ***@Method***, los usuarios también deben proporcionar la clase de declaración y la descripción de los parámetros. Por un lado, la clase de declaración de una función es la clase que contiene la implementación de la tarea y es necesaria para vincular la tarea con la implementación del método. Por otro lado, la descripción de los parámetros se indica agregando una anotación ***@Parameter*** a cada parámetro de la tarea. Esta anotación es necesaria para construir el gráfico de dependencia de las tareas. El contenido obligatorio de la anotación de parámetro es el tipo (que debe referirse a cualquier tipo básico de Java, una cadena, un objeto o un archivo) y la dirección (donde los únicos valores válidos son *IN*, *OUT* e *INOUT*). En la Figura 1.4 proporcionamos el código de la clase «***IncrementImpl.java***».

```java
package increment;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.FileNotFoundException;


public class IncrementImpl {

    public static void increment(String counterFile) throws FileNotFoundException, IOException {
        // Read value
        System.out.println("***** File :" + counterFile);
        FileInputStream fis = new FileInputStream(counterFile);
        int count = fis.read();
        fis.close();

        // Write new value
        FileOutputStream fos = new FileOutputStream(counterFile);
        fos.write(++count);
        fos.close();
    }
```

Figura 1.4: Increment Implentation class

A continuación compilaremos y ejecutaremos la aplicación Incremento con el comando runcompss y la opción ***-g*** para poder generar el grafo al final de la ejecución. En la Figura 1.5 vemos los comandos para compilar y ejecutar la aplicación y también la salida de la ejecución de *Increment*.

```
compss@bsc:~$ cd ~/tutorial_apps/java/increment/src/main/java/increment/
compss@bsc:~/tutorial_apps/java/increment/src/main/java/increment$ javac *.java
compss@bsc:~/tutorial_apps/java/increment/src/main/java/increment$ cd ..
compss@bsc:~/tutorial_apps/java/increment/src/main/java$ jar cf increment.jar increment
compss@bsc:~/tutorial_apps/java/increment/src/main/java$ mv increment.jar ~/tutorial_apps/java/increment/jar/

compss@bsc:~$ cd ~/tutorial_apps/java/increment/jar
compss@bsc:~/tutorial_apps/java/increment/jar$ runcompss -g increment.Increment 10 1 2 3
[  INFO] Using default execution type: compss
[  INFO] Using default location for project file: /opt/COMPSs/Runtime/configuration/xml/projects/default_project.xml
[  INFO] Using default location for resources file: /opt/COMPSs/Runtime/configuration/xml/resources/default_resources.xml

---------- Executing increment.Increment -------------------------

WARNING: COMPSs Properties file is null. Setting default values
[(1028)    API]  -  Starting COMPSs Runtime v<version>
Initial counter values:
- Counter1 value is 1
- Counter2 value is 2
- Counter3 value is 3
Final counter values:
- Counter1 value is 11
- Counter2 value is 12
- Counter3 value is 13
[(4403)    API]  -  Execution Finished

------------------------------------------------------------
```

Figura 1.5: Ejemplo de ejecución de Increment

En la Figura 1.6 proporcionamos el grafo de ejecución de las tareas de la aplicación *Increment*.

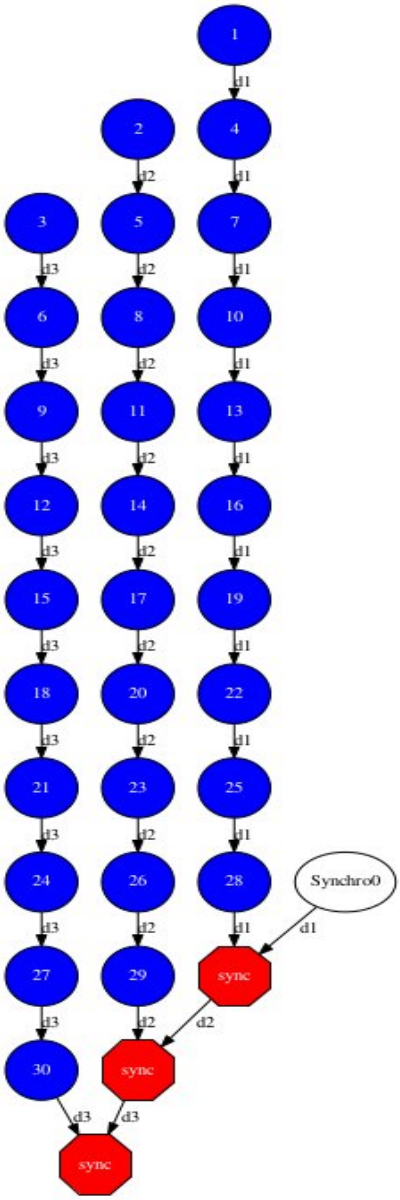


Figura 1.6: Java increment tasks graph

El comando runcompss tiene varios argumentos de línea de comandos (que se detallan al ejecutar ***runcompss –help*** en la consola), en la Tabla 1.1 se proporciona una breve descripción de algunos que son más útiles para el desarrollo.

Argumento	Descripción
-d	Habilita el modo debug
-g	Habilitan la generación del grafo de dependencias de las tareas
-m	Habilita la herramienta de monitoreo
-t	Habilita la herramienta de tracing
- -sumary	Proporciona un resumen de las tareas al final de la ejecución
- -lang=<str>	Habilita el binding de python y C/C++
- -project=<str>	Establece un fichero de configuración específico
- -resources=<str>	Establece un fichero de recursos específico
- -classpath=<str>	Agrega un classpath específico al entorno de ejecución

Tabla 1.1: Argumentos para el comando runcompss

Runtime System

Para poder abstraer las aplicaciones de la infraestructura subyacente, COMPSs se basa en su Runtime System que generara un proceso Master en la máquina donde se ejecuta la aplicación y uno o varios procesos Worker donde se lanzarán las tareas(ver la Figura 1.7). Estos procesos se comunican a través de la red y pueden enviarse mensajes entre sí para organizar la ejecución y paralelización de las tareas de la aplicación.

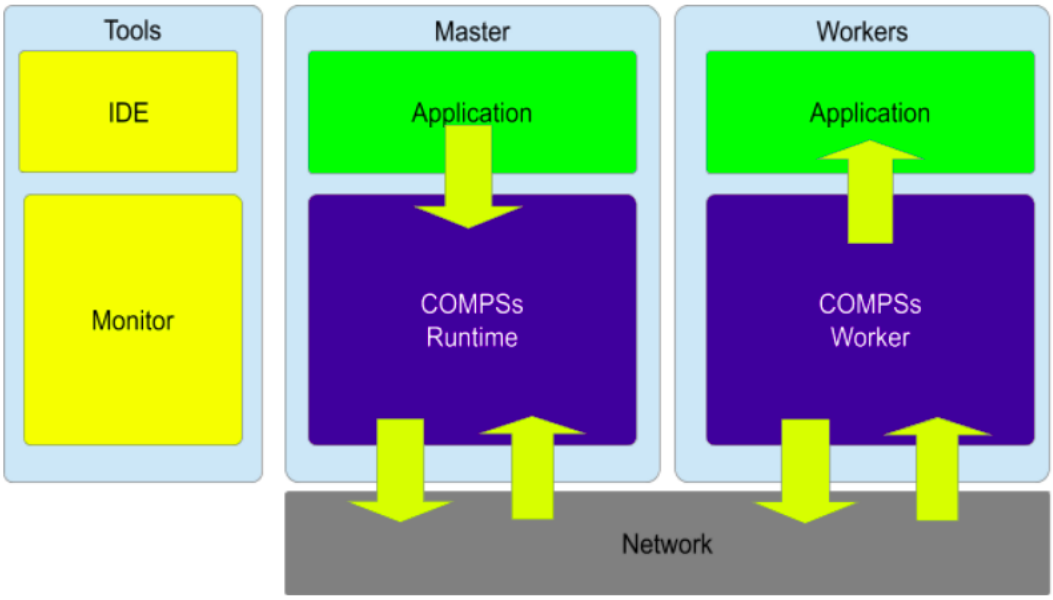


Figura 1.7: Estructura de COMPS

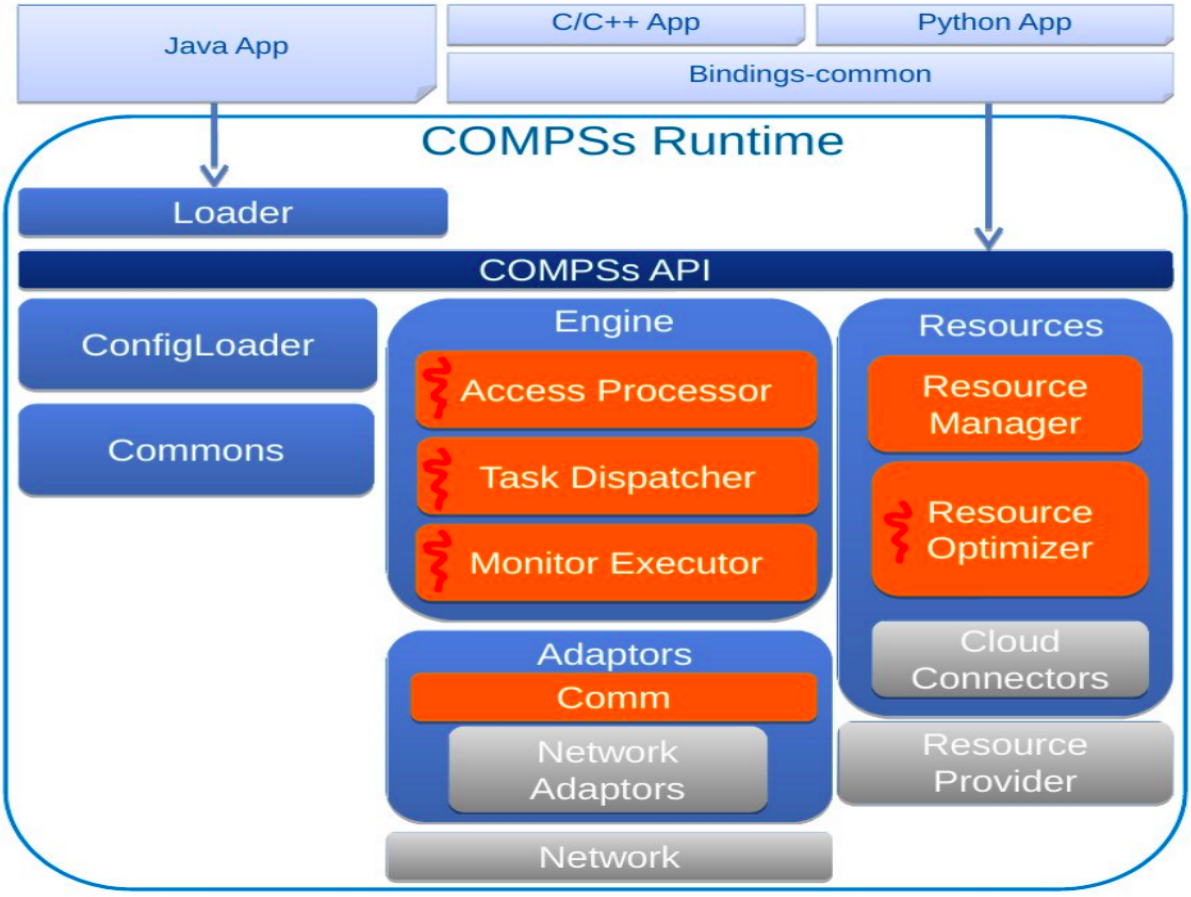


Figura 1.8: Runtime de COMPSs

En la figura 1.8 se muestra como es la interacción entre la aplicación y el *Runtime* en los diferentes lenguajes que soporta. Una vez que se inicia una aplicación de Java, el *Runtime* activa un *Java ClassLoader* que usa *Javassist* [9] para instrumentar la clase principal de la aplicación. La instrumentación modifica el código original insertando las llamadas necesarias a la interfície de COMPSs para generar las tareas, manejar las dependencias y agregar sincronizaciones de datos. Para lograr el mismo propósito en las aplicaciones de Python, el binding de Python (PyCOMPSs) [5] analiza los decoradores del código principal y agrega las llamadas necesarias a la interfície de COMPSs. En el caso de las aplicaciones C/C++, COMPSs también requiere un archivo de interfaz que se utiliza al compilar la aplicación para generar índices para el código principal, agregar las llamadas API de COMPSs necesarias y generar el código para la ejecución de tareas en los recursos.

Podemos identificar cinco componentes importantes en el Runtime de COMPSs, estos son:

- **Commons:** Contiene los elementos comunes utilizados por todos los componentes del Runtime.
- **ConfigLoader:** Carga el proyecto entero con los archivos de configuración (proyecto y recursos) y los parámetros de configuración de *JVM*[5].
- **Engine:** Contiene los submódulos para manejar la detección de tareas, las dependencias de datos y la gestión de tareas. El *Access Processor* monitorea los accesos de datos para que el Runtime pueda

[5]Java Virtual Machine

construir las dependencias de datos entre tareas, el *Task Dispatcher* controla el ciclo de vida de las tareas y el *Monitor Executor* las estructuras para el monitoreo en tiempo real.

- **Resources:** Maneja todos los recursos disponibles en la infraestructura subyacente. Este componente crea, destruye y monitorea el estado de todos los recursos disponibles. Dado que COMPS admite la elasticidad a través de conectores en la nube, este componente contiene un *Resource Optimizer* que se encarga de crear y destruir recursos.
- **Adaptors:** Contiene las diferentes implementaciones de adaptadores de comunicación. Esta capa se utiliza para comunicar al Master y los Workers, y abstrae al Runtime de los diferentes adaptadores de red.

Flujo de ejecución de las tareas

Para detallar como funciona COMPSs al ejecutar una aplicación, la Figura 1.9 describe el ciclo de vida de las tareas. Desde el código principal de la aplicación, la interfície registra todas las tareas. Teniendo en cuenta las tareas registradas, el Runtime crea un grafo basado en las dependencias de datos. Después, este grafo se envía al *Task Dispatcher* que se encarga de programar las tareas libres de datos cuando se liberan de las dependencias. Esto significa que una tarea solo se programa cuando no tiene dependencia de datos y hay suficientes recursos libres para ejecutarla (cada tarea puede tener restricciones diferentes y, por lo tanto, no se programa si no hay un recurso que cumnpla con los requisitos).

Finalmente, se puede programar una tarea y, luego, se envía a ejecutar. Este paso incluye la creación del Job, la transferencia de los datos de entrada al recurso seleccionado, la ejecución de la tarea real en el Worker y la recuperación de salida hacia el Master. Si alguno de estos pasos falla, COMPS proporciona mecanismos de tolerancia a fallos para re-programar tareas.

Una vez que la tarea ha finalizado, el Runtime almacena los datos de monitoreo de la tarea, sincroniza los datos requeridos por la aplicación y libera las tareas dependientes para que puedan programarse.

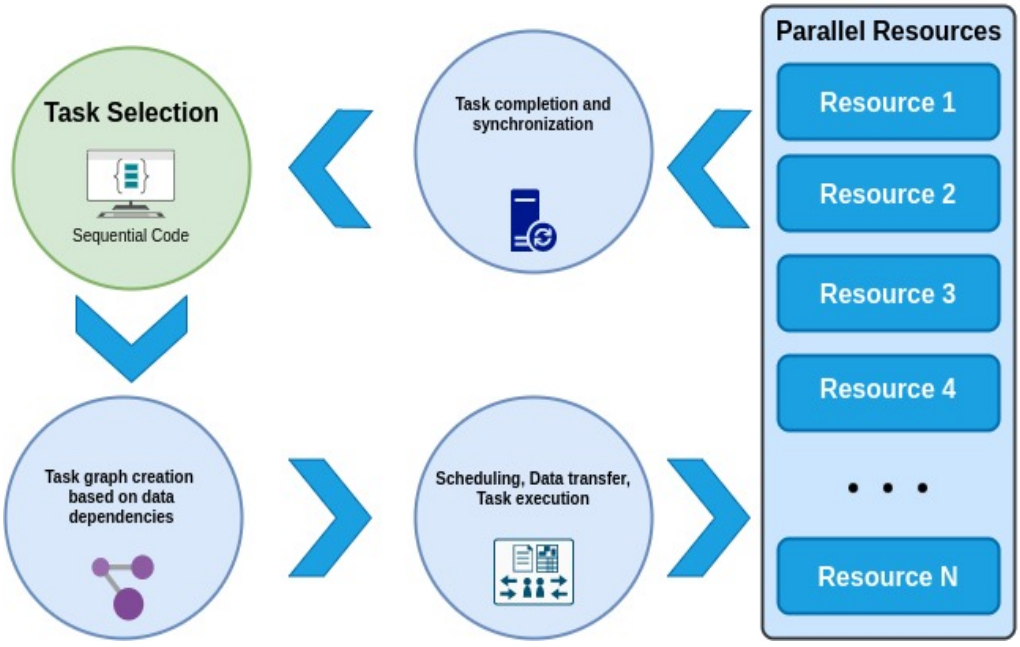


Figura 1.9: Flujo de ejecución de las tareas

1.2.2. Contenedores

Un contenedor [10] tiene dos estados: *pasivo o en ejecución*. Cuando está pasivo, es un fichero (o conjunto de ficheros) que se guarda en el disco. Cuando ejecutamos un comando para iniciar un contenedor, la plataforma del contenedor desempaqueta los archivos y metadatos requeridos para entregarlos al kernel de Linux para iniciar su ejecución. Iniciar un contenedor es muy similar a iniciar un proceso normal de Linux, requiere hacer una llamada API al kernel, esta llamada generalmente inicia el aislamiento de recursos del kernel y monta una copia de los archivos que estaban en la imagen al contenedor. Una vez que se ejecutan, los contenedores son solo un proceso de Linux y pasan a estar en estado de ejecución.

El proceso para iniciar contenedores, así como los formatos de imagenes, están definidos y regidos por diferentes estándares. En la actualidad la industria de los contenedores está avanzando con un estándar regido por la *Open Container Initiative* [11], también denominado OCI. El alcance de OCI incluye una especificación que define el formato en disco de imágenes para contenedores, así como los metadatos que definen cosas como la arquitectura de hardware y el sistema operativo (Linux, Windows, etc.). Tener un formato de imagen de contenedores para toda la industria permite que florezcan diferentes ecosistemas de software, contribuyentes, proyectos y proveedores que pueden aportar herramientas e imágenes que sean totalmente compatibles entre ellas. Por otro lado, la forma en que una imagen se convierte en un contenedor también está estandarizado por la OCI, que incluye una especificación llamada *RunC* [12][6]. Esta implementación de referencia es de código abierto y es comúnmente utilizada por muchas plataformas de contenedores para comunicarse con el núcleo del host al crear contenedores.

Las herramientas orientadas a la especificación OCI, *Container Image Format Specification* [13][7] y *Container Runtime Specification* [14][8] garantizan la portabilidad entre un amplio ecosistema de plataformas de contenedores y herramientas de soporte entre proveedores de la nube y arquitecturas. Comprender la nomenclatura, los estándares y la arquitectura de contenedores nos garantiza poder comunicarnos con otros desarrolladores para crear aplicaciones y entornos contenerizados escalables y compatibles.

Cabe destacar que Docker y Singularity siguen el estándar OCI, lo que hace que los formatos de imagen de ambas plataformas sean totalmente compatibles entre ellas. A continuación pasaremos a explicar brevemente las plataformas Docker y Singularity, que han sido solicitadas específicamente en los requisitos para la integración dentro de COMPSs.

[6] RunC es una herramienta CLI para generar y ejecutar contenedores de acuerdo con la especificación OCI.

[7] Container Image Format Specification crea y mantiene la especificación de formato de imagenes de contenedores

[8] Container Runtime Specification desarrolla especificaciones para estándares sobre procesos de SO's y contenedores de aplicaciones

Docker

Docker [15] es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones en contenedores. Este contenedor empaqueta todo lo necesario para que uno o más procesos funcionen. Esto garantiza que siempre se podrá ejecutar, independientemente del entorno en el que queramos desplegarlo. No hay que preocuparse de qué software ni versiones tiene nuestra máquina, ya que nuestra aplicación se ejecutará dentro del contenedor.

Existen tres componentes específicos de docker que nos interesa mencionar, estos son:

- **El Docker *daemon* o Doker Engine:** Se trata de un capa entre los contenedores y el kernel de Linux, es el runtime que administra los contenedores y es independiente del sistema operativo subyacente.
- **Dockerfile:** Es un documento de texto que contiene toda la información de configuración y los comandos necesarios para ensamblar la imagen de lo que se convertirá en un *contenedor*.
- **API REST:** Especifica las interfaces que los programas pueden usar para comunicarse con el *daemon* e indicarle qué hacer.
- **CLI:** Una interfaz de línea de comandos (el comando docker).

En resumen, la CLI utiliza la API REST de Docker para interactuar con el *daemon* través de scripts o comandos directos de la CLI. Docker Engine [16] es una aplicación cliente-servidor que gestiona toda la ejecución de los contenedores, en la Figura 1.10 podemos ver todos los componentes mencionados anteriormente.

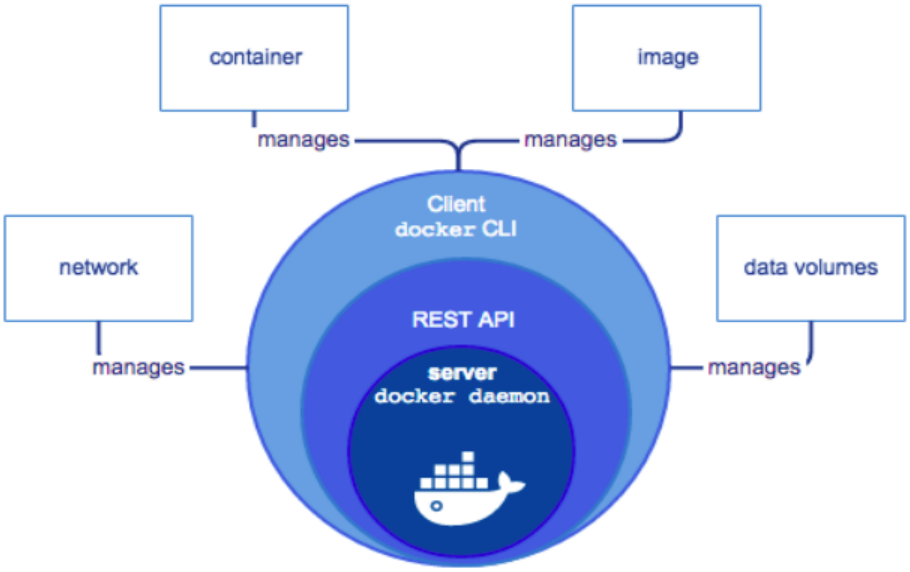


Figura 1.10: Docker Engine

Singularity

Singularity [17] es una plataforma de contenedores que nos permite crear y ejecutar contenedores que empaquetan piezas de software de forma portable y reproducible. Podemos construir un contenedor usando Singularity en nuestro ordenador portátil y luego ejecutarlo en la nube o en cualquier entorno distribuido. En Singularity, un contenedor es un simple fichero y además de ser compatible con las imágenes de Docker, no tiene un proceso daemon para su ejecución, tampoco tiene cambios contextuales de usuario ni escalado de permisos; el usuario dentro del contenedor siempre es el mismo usuario que inicio el contenedor.

En la figura 1.11 podemos ver que un contenedor Singularity utiliza directamente los usuarios del anfitrion para su ejecución y que también, tanto dentro del contenedor como fuera de él, puede usar la misma distribución y versión de la interfaz de paso de mensajes MPI [18].

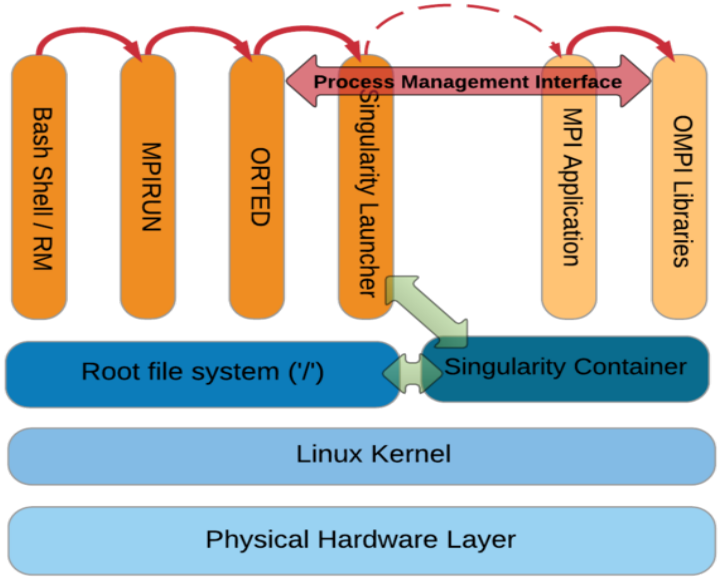


Figura 1.11: Interacción MPI Contenedor-Anfitrión

1.3. Alcance del proyecto

En esta sección daré una descripción del alcance de nuestro proyecto. Describiré primero los requerimientos solicitados, el objetivo general y después acabaré explicando la metodología a seguir y los posibles obstáculos.

1.3.1. Requerimientos

Los requerimientos especificados por los desarrolladores de COMPSs son los siguientes:

- Se requiere que el usuario, de forma fácil y transparente, pueda informar al runtime los parámetros indispensables para ejecutar binarios dentro de software contenerizado que se haya seleccionado.
- Integrar en el runtime los motores de contenedores Docker y Singularity.
- Las modificaciones del runtime y la interficie, no deben romper las versiones anteriores y deben ajustarse a la línea base que se esta desarrollando.
- Cualquier cambio o petición que se dé durante el proyecto.

1.3.2. Objetivos

El objetivo general de este proyecto es el desarrollo de un prototipo, que permita a COMPSs gestionar la combinación de software contenerizado, para permitir la integración de diferentes piezas de software en una sola aplicación.

El escenario en que nos encontramos cuando queremos ejecutar tareas con binarios en diferentes recursos, es que no siempre tenemos instaladas las librerías necesarías para ejecutar estos binarios o simplemente estas librerías colisionan cuando hay varias versiones. Una solución que proponemos es ejecutar las tareas dentro de contenedores para evitar tener que instalar dependencias en los recursos, simplemente hacer que COMPSs gestione la ejecución de una tarea en un contenedor como una pieza de software independiente, donde el usuario, pueda informar a la interficie los parámetros para contenerizar la tarea. Estos parámetros tienen que ser los indispensables, por ejemplo, informar la imagen donde se ejecutará el binario y el motor de contenerización que cargará la imagen.

Los siguientes puntos resumen los objetivos principales de este proyecto:

1. Investigar las características y funcionamiento de Docker y Singularity para la implantación de la nueva funcionalidad dentro de COMPSs y cómo podrían interactuar con la interfície.
2. Crear nuevas anotaciones en la interficie de COMPSs para que los usuarios puedan informar los parámetros de contenerización para ejecutar las tareas binarias; extender las anotaciones para la ejecución de tareas binarias ya existentes, manteniendo la compatibilidad con versiones anteriores.
3. Validadar el funcionamiento del desarrollo utilizando una aplicación real en el entorno de pruebas de COMPSs.

1.3.3. Metodología y rigor

A continuación se describe la metodología y las herramientas de trabajo que hemos seguido para realizar el proyecto.

Método de trabajo

En el desarrollo del proyecto utilizaremos una metodología en cascada de cuatro fases. La primera fase será la de gestión del proyecto, donde se preparará el proyecto definiendo el contexto, el alcance, objetivos y la planificación temporal y económica.

En la segunda fase se seguirá una metodología iterativa. Se diseñarán pequeñas partes del prototipo que se implementarán y se probarán continuamente, de este modo se prioriza el correcto funcionamiento de las tareas más pequeñas, que luego se convertirán en el proyecto. En la tercera fase, se realizarán las validaciones finales del desarrollo en entornos de pruebas.

Finalmente, la última fase constará de la redacción final de la memoria y la preparación de la defensa del proyecto. En la Figura 1.12 podemos ver las fases por las que pasa el proyecto.

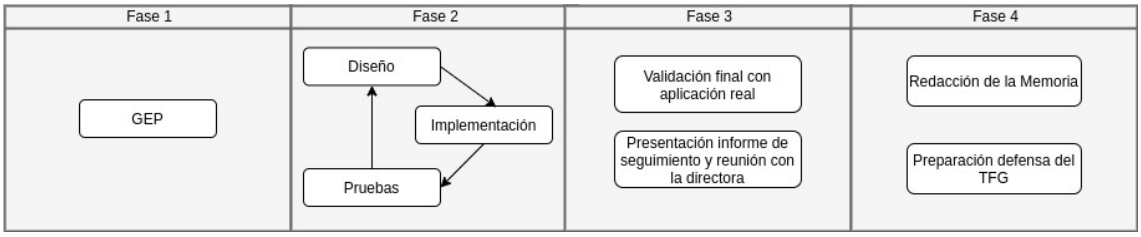


Figura 1.12: Fases del proyecto

Herramientas de seguimiento

Para el seguimiento del proyecto utilizaremos la herramienta Trello [19], además de planificar reuniones semanales con el codirector. Para manejar el código del proyecto, utilizaremos el sistema control de versiones git con el gestor de repositorios GitLab y se creará una rama dentro del proyecto de COMPSs para realizar el desarrollo. La comunicación con la directora y el codirector del proyecto se realizará de manera presencial, telemática por Skype y por correo electrónico.

Metodología de validación

Para validar los avances que se van produciendo a lo largo de la realización del proyecto, se utilizarán los siguientes métodos de validación:

- **Reuniones semanales de seguimiento:** Se realizarán reuniones cada semana con el codirector para comprobar y validar que la planificación temporal y el desarrollo del proyecto cumple con los planes marcados y poder solucionar problemas que vayan surgiendo durante el desarrollo.
- **Validación final:** Para finalizar, se realizarán ensayos con una aplicación real para validar que la funcionalidad nueva es aceptable y funciona correctamente en el entorno de pruebas de COMPSs.

1.3.4. Posibles obstáculos

Los condicionantes más destacables a los que se deben hacer frente durante el desarrollo se han agrupado de la siguiente manera:

Restricción del tiempo

Disponemos de un cuatrimestre para el desarrollo, lo que supone un tiempo extremadamente limitado. Por este motivo tenemos que realizar una planificación muy detallada y dejar margen de flexibilidad temporal para evitar posibles imprevistos. Hemos estimado terminar la fase de implementación tres semanas antes de la entrega final para poder ocupar este tiempo en caso de desviaciones en las tareas.

Curva de aprendizaje

La falta de conocimiento sobre la arquitectura de COMPSs, contenerización y herramientas para su desarrollo es un obstáculo, ya que debemos invertir una gran cantidad de tiempo en su aprendizaje, el cual se hará de manera autónoma. La adaptación al entorno de desarrollo de COMPSs y su particularidades conlleva un tiempo considerable.

Trabajar sobre un código existente

Ya que este proyecto se basa en realizar la integración, de una funcionalidad nueva en un framework con varias versiones, debo moverme en un código realizado por otras personas y no romper la compatibilidad con las versiones anteriores.

2

Integración de Docker y Singularity en COMPSs

Para realizar la integración en COMPSs necesitaremos, por una parte investigar el funcionamiento y características de la interficíe de Docker y Singularity para que soporte las características del Runtime, conocer bien el flujo y la estructura interna de ejecución de binarios dentro de COMPSs, así como el proceso habitual de desarrollo de una aplicación que utiliza la interficie que modificaremos. Para realizar las pruebas hemos utilizado la versión de COMPSs 2.6, la versión de *Docker* 19.03.7 y la versión de *Singularity* 3.5.2.

2.1. Estudio de la interficie de Docker y Singularity para la integración

En este apartado vamos a investigar la ejecución de binarios dentro de Docker y Singularity. Tenemos que recopilar información sobre cómo se ejecutan comandos en ambas plataformas e identificar las características que nos interesan. Entre estas características tenemos, no dejar instancias de contenedores abiertas en el recurso que ejecuta la pieza de software contenerizada, utilizar directorios comunes o compartidos para no interferir en el mecanismo de transporte de datos entre el Master y los Workers, y por último, soportar descriptores de ficheros.

2.1.1. Comando *docker run*

Para la ejecución de binarios con contenedores Docker utilizaremos *docker run*, que crea y arranca un contenedor para ejecutar unos comandos ya establecidos en el fichero Dockerfile y/o los que se pasen como parámetros en el comando. De estos, sólo tenemos un parámetro obligatorio que es el nombre de la imagen que usará para crear el contenedor. Entre las opciones que debemos pasar al comando, debemos pensar en las características mínimas que requiere el runtime, estas las explicaremos en el siguiente punto, el comando y los argumentos de la figura 2.1, serían el binario y los argumentos que el programador quiere ejecutar dentro del contenedor.

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Figura 2.1: Uso del comando *docker run*

Ahora vamos a describir las opciones que utilizaremos para ejecución de contenedores con el comando *docker run*, hemos elegido las siguientes opciones pensando en las características necesarias para realizar una buena integración en el framework.

Opción CLEAN UP [- *-rm*] [20]

Durante las pruebas que realizamos con Docker, notamos que al ejecutar *docker run*, se creaban instancias de contenedores que no se destruían automáticamente, llenando el sistema host de instancias innecesarias. Una opción que se consideró en su momento fue utilizar el comando *docker exec*, el cual necesitaba tener una instancia del contenedor abierta en el host permanentemente, pero fue descartada por no cumplir con los requerimientos del proyecto. Considerando que necesitábamos el contenedor solamente para ejecutar un binario, concluimos que la opción - *-rm* elimina automáticamente el contenedor cuando terminamos de ejecutar el binario seleccionado, por lo que fue seleccionada para ser utilizada.

Uso de bind mounts y volumenes [21] en Docker

Un volumen es un directorio o un fichero en el *docker engine* que se monta directamente en el contenedor. Podemos montar varios volúmenes en un contenedor y en varios contenedores podemos montar un mismo volumen.

Existen tres tipos de almacenamiento en *Docker*:

- **volumenes:** Docker almacena los datos dentro de un área que él controla del sistema de ficheros. Es el mecanismo preferido para persistir los datos en los contenedores y el que utilizaremos para la integración. Los volúmenes se almacenarán en el directorio **/var/lib/docker/volumes/** y solo Docker tiene permisos sobre esta ubicación. Para esta opción se utiliza el flag *-v* seguido del directorio de la máquina host que vamos a compartir y el directorio dentro del contenedor donde se montará, entre ambos paths dos puntos(*:*). La opción completa será de esta forma:
  -*v* path_directorio_host:path_directorio_contenedor .
- **bind mounts:** Se utiliza para mapear cualquier sitio del sistema de ficheros dentro de tu contenedor. A diferencia de los volúmenes, a través de este mecanismo es posible acceder a la ruta mapeada y modificar los ficheros.
- **tmpfs:** Se trata de un almacenamiento temporal en memoria. Se suele utilizar para el almacenamiento de configuraciones y espacios efímeros que desparecerán cada vez que el contenedor se pare.

En la figura 2.2 podemos ver los tres tipos de almacenamiento que hemos mencionado anteriormente, volumenes, bind mount y tmpfs.

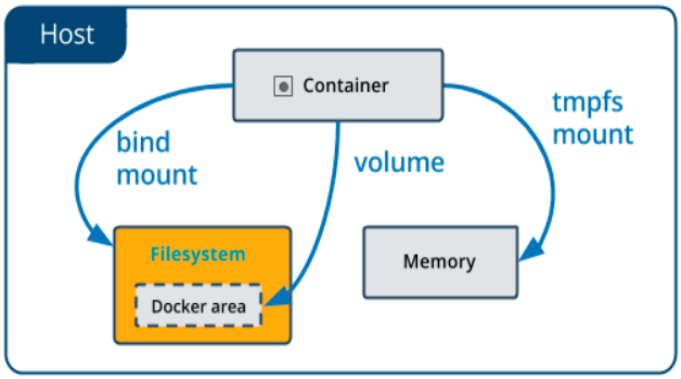


Figura 2.2: Almacenamiento en Docker

Opción WORKDIR *-w* [22]

Por defecto, el directorio de ejecución de binarios dentro de un contenedor es **/**, pero podemos seleccionar el directorio que necesitemos para la ejecución de los binarios. Esta opción nos viene muy bien ya que el Runtime de COMPSs ejecuta los binarios desde el directorio de ejecución que genera para poder sincronizar los datos entre el nodo Master y el Worker. El operador para esta opción es ***-w path_working_directory***.

Opción para activar descriptor de ficheros de entrada [23]

Durante las pruebas, hemos concluido que el comando *docker run* no nos conectaba automáticamente al *standar input* (*STDIN*). Para enlazar el *standar input* del Runtime al *standar input* del proceso que se ejecuta en el contenedor, es necesario utilizar la opción ***-i*** para conectarnos. Tenemos muchas mas opciones para conectarnos al standar input en Docker, pero la opción ***-i*** es compatible con el Runtime de COMPSs y nos viene perfecto para la integración.

Resumen de las opciones del *docker run*

En la Tabla 2.1 hacemos un resumen de las opciones que utilizaremos durante la integración.

Opciones	Descripción
- -r	Remueve el contenedor automáticamente después de ejecutar *docker run*
- v path_host:path_container	Monta un volumen compartido entre el host y el contenedor
- w path_working_directory	Define el directorio de ejecución dentro del contenedor
- i	Mantiene la consola de entrada *STDIN* abierta, incluso si no esta conectada

Tabla 2.1: Opciones de *docker run* para la integración

2.1.2. Comando *singularity exec*

Para la ejecución de binarios en contenedores Singularity utilizaremos *singularity exec*, que nos permitirá ejecutar binarios dentro de un contenedor especificando el archivo de la imagen, *singularity exec* crea un contenedor efímero que ejecuta el comando y desaparece automáticamente. Al igual que *docker run*, tenemos un parámetro obligatorio que es el path completo de la imagen que se usará para crear el contenedor. En el siguiente punto explicaremos las opciones que debemos pasar a *singularity exec* para la integración. En la figura 2.3,podemos ver los parámetros necesarios para ejecutar el comando *singularity exec*.

```
singularity exec [exec options...] <container> <command>
```

Figura 2.3: Uso del comando *singularity exec*

Uso de bind paths y montaje [24] en Singularity

Singularity nos permite asignar directorios del sistema host dentro del contenedor mediante enlace a directorios, esto nos permitirá leer y escribir datos en el sistema host con facilidad. Podemos habilitar esta opción utilizando el flag *–bind* y especificando las rutas del enlace como lo hacíamos en DOcker. En la Figura 2.4 podemos ver un ejemplo del uso de *–bind* enlazando en directorio ***/data*** del host en el directorio ***/mnt*** del contenedor.

```
$ ls /data
bar  foo

$ singularity exec --bind /data:/mnt my_container.sif ls /mnt
bar  foo
```

Figura 2.4: Uso de bind paths

2.1.3. Comparación

Durante las pruebas en ambas plataformas hemos notado la facilidad de uso de singularity frente a Docker, podríamos destacar que en singularity el usuario que inicia el contenedor en el host, es el mismo dentro del contenedor ahorrandonos tener que definir un directorio de ejecución y escalar usuarios, en singularity el contenedor desaparece automáticamente después de ejecutar un binario y no tuvimos el *standar input (STDIN)* ya que al utilizar el mismo usuario del host dentro del contenedor no es necesario. El usuario tiene la facilidad de convertir imágenes públicas de Docker en el formato nativo de Singularity, es decir, el formato de imagen de Singularity (*SIF*). Por último, ambas plataformas comparten de manera similar el enlace de directorios en los contenedores lo que nos ha facilitado continuar rápidamente con el proyecto.

2.2. Integración

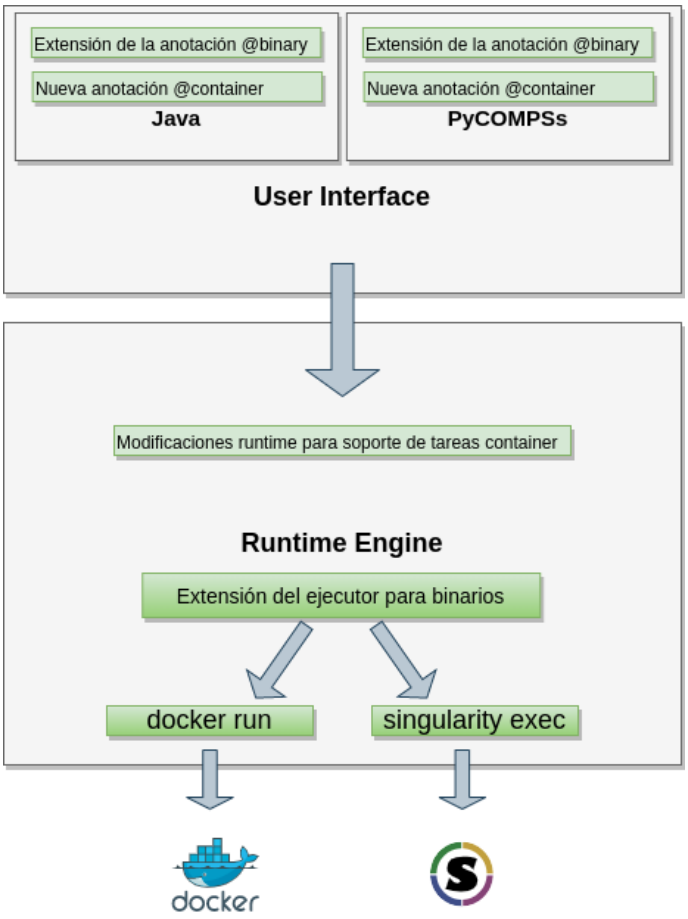


Figura 2.5: Visión general de la integración

Como vimos en la visión general de COMPSs, el modelo de programación que seguimos define anotaciones que deben agregarse al código secuencial para ejecutar las tareas en paralelo. Podemos dividir estas anotaciones en dos grupos:

- **Anotaciones de métodos:** Anotaciones agregadas a los métodos de código secuencial para poder detectarlas como tareas y potencialmente ejecutarlas en paralelo.
- **Anotaciones de parámetros:** Anotaciones agregadas a los parámetros de los métodos para manejar las dependencias y transferencias de datos.

Para realizar la integración, añadiremos una nueva anotación de métodos que llamaremos ***@Container***. Esta anotación la añadiremos en la interficie de Java y en los binding de Pyhton (*PyCOMPSs* [5]), que es donde el usuario informará al Runtime los parámetros para la contenerización del binario. Por otra parte, extenderemos la anotación de método ***@Binary*** para que el usuario pueda informar los nuevos parámetros de la anotación para contenerizar el binario, para esta extención también modificaremos la interficie de Java y los binding de Pyhton. En el runtime realizaremos modificaciones en el flujo de ejecución de binarios para soportar los nuevos parámetros de las extensiones y la nueva anotación, también modificaremos los ejecutores de binarios añadiendo los comandos ***docker run*** y ***singularity exec*** para

que estos se puedan ejecutar contenerizados en *DOCKER* o *SINGULARITY*, o sin contenerización. En la Figura 2.5, podemos ver la visión general de los cambios que se realizaron durante la integración.

Durante las fases de diseño hemos estudiado la información que debia pasar el programador a la interfície, consideramos que ninguna de las opciones de los comandos ***docker run*** y ***singularity exec*** eran necesarias. Únicamente se necesita el nombre de la plataforma y el nombre de la imagen, con esta información el programador ya es capaz de contenerizar binarios en sus tareas.

2.2.1. Modificaciones en la anotación *@Binary*

Extendemos dos parámetros en la anotación ***@binary*** que tanto para *Python* como *Java* serán identicos. Estos parámetros se utilizarán para seleccionar el tipo de contenedor y la imagen donde se ejecutará la tarea del binario.

Los nuevos parámetros de la anotación ***@binary*** para la interfície de *Python* [25] y *Java* [26] son:

- **engine:** String que indica el tipo de contenedor que vamos a utilizar para ejecutar el binario, los tipos son Docker o Singularity.
- **image:** String que indica el nombre de la imagen, en el caso de *SINGULARITY*, vamos a utilizar el path completo donde se encuentre la imagen.

En la Figura 2.6 podemos ver un ejemplo de uso de ***@binary*** para *Python*.

```
@binary(binary="ls", engine='DOCKER', image='ubuntu')
@task(result={Type:FILE_OUT, StdIOStream:STDOUT})
def execLS(result):
    pass

@binary(binary="pwd", engine='SINGULARITY', image='/home/compss/singularity/examples/ubuntu_latest.sif')
@task(result={Type:FILE_OUT, StdIOStream:STDOUT})
def execPWD(result):
    pass
```

Figura 2.6: Anotación ***@binary*** de *Python*

En la Figura 2.7 podemos ver un ejemplo de uso de ***@Binary*** para *Java*.

```
@Binary(binary="ls", workingDir="/home/compss/",
                        container = @BinaryContainer(engine="SINGULARITY", image="/home/compss/singularity/examples/ubuntu_latest.sif"))
void execLS(
        @Parameter(type = Type.FILE, direction = Direction.OUT, stream = StdIOStream.STDOUT) String out
        );


@Binary(binary="ls", container = @BinaryContainer(engine="DOCKER", image="centos"))
void execPWD(
        @Parameter(type = Type.FILE, direction = Direction.OUT, stream = StdIOStream.STDOUT) String out
        );
```

Figura 2.7: Anotación ***@Binary*** de *Java*

2.2.2. Nueva anotación *@Container*

La nueva anotación ***@container*** sirve para ejecutar la tarea de un binario dentro de un contenedor con una imagen seleccionada por el programador a través de los parámetros de esta interfície.

Los parámetros de la nueva anotación ***@container*** para la interfaz de *Python* y *Java* quedan de la siguiente manera:

- **engine:** String que indica el tipo de contenedor que vamos a utilizar para ejecutar el binario, los tipos son *Docker o Singularity*.
- **image:** String que indica el nombre de la imagen, en el caso de *SINGULARITY*, vamos a utilizar el path completo donde se encuentre la imagen.
- **binary:** String que define el binarios que se quiere ejecutar..
- **working_dir:** String con el path completo del directorio de trabajo dentro del nodo de COMPSs.

En la Figura 2.8 podemos ver un ejemplo de uso de ***@container*** para *Python.*

```
@container(engine="DOCKER", image="centos", binary="/home/compss/exec", working_dir="/home/compss/")
@task(result={Type:FILE_OUT, StdIOStream:STDOUT})
def binary_func(result):
    pass

@container(engine="SINGULARITY", image="/home/compss/singularity/examples/ubuntu_latest.sif",
                                                    binary="ls", working_dir="/home/compss/")
@task(result={Type:FILE_OUT, StdIOStream:STDOUT})
def containerExec(result):
    pass
```

Figura 2.8: Anotación ***@container*** de *Python*

En la Figura 2.9 podemos ver un ejemplo de uso de ***@Container*** para *Java.*

```
@Container(engine="DOCKER", image="centos", binary="ls", workingDir="/test")
void lsContainer(
        @Parameter(type = Type.FILE, direction = Direction.OUT, stream = StdIOStream.STDOUT) String out
        );

@Container(engine="SINGULARITY", image="/home/compss/singularity/examples/ubuntu_latest.sif",
                                                    binary="/test/ejecutable/exec", workingDir="/test")
void pwdContainer(
        @Parameter(type = Type.FILE, direction = Direction.OUT, stream = StdIOStream.STDOUT) String out
        );
```

Figura 2.9: Anotación ***@Container*** de *Java*

2.2.3. Modificaciones en el *Runtime*

Para identificar claramente las modificaciones que debíamos realizar en el *runtime*, hemos realizado pruebas en local utilizando métodos con la anotación ***@Binary*** para conocer el flujo de ejecución de las tareas binarias dentro del *runtime*. Cuando teníamos claro el proceso de compilación y ejecución de aplicaciones en COMPSs, pasamos a realizar las modificaciones en el código del framework. Ya que el objetivo de la nueva anotación es ejecutar un binario dentro de un contenedor, hemos decidido empezar con la extención de parámetros en ***@Binary*** para tener claro el flujo de ejecución e identificar los puntos donde había que modificar el código.

Para modificar la anotación ***@Binary***, primero hemos incorporado y realizado cambios en el proyecto ***compss-api*** del framework para definir todos los nuevos parámetros de entrada que tendría esta anotación. Posteriormente, fuimos incorporando en los proyectos del framework los cambios necesarios para extenderlos. Para la integración de los comandos *docker run* y *singularity exec*, estos fueron incorporados en la clase *BinaryInvoker.java* del proyecto ***compss-adaptor-execution*** ya que es el punto donde se preparan los comandos para ser ejecutados en los Workers. La clase *BinaryInvoker.java*, que se encarga de preparar el comando que se lanzará en el Worker con los parámetros introducidos en la anotación ***@Binary*** por el usuario.

Cuando el usuario informe al método con los parámetros de contenerización (engine e imagen), el binario se ejecutará contenerizado, en cambio si no informamos esto, el binario se ejecuta tal cual en las anteriores versiones del framework, respetando así la implementación anterior. En la Tabla 2.2 tenemos los comandos ***docker run*** y ***singularity exec*** de *DOCKER y SINGULARITY* que utilizaremos para la integración en COMPSs.

Engine	Comando
DOCKER	docker run -i –rm -v [pathHostDir]:[pathContainerDir] -w [pathHostDir] [image] [binary] [args. . .]
SINGULARITY	singularity exec –bind [pathHostDir]:[pathContainerDir] [image] [binary] [args. . .]

Tabla 2.2: Comandos *docker run* y *singularity exec* para integrar en la clase *BinaryInvoker.java*

Una vez modificada la anotación ***@Binary***, hemos continuado con la creación de la nueva anotación ***@Container***. Hemos creado las clases necesarias en la interfície y el *runtime engine* del framework para la nueva anotación siguiendo el modelo del flujo de ejecución de ***@Binary***. Practicamente hemos utilizado el mismo flujo para respetar las versiones anteriores, por tiempo nos hemos limitado a realizarlo de esta manera, pero se podría crear un flujo nuevo en nuevas versiones del framework. En la tabla 2.3 tenemos un resumen de las clases que hemos añadido y modificado en la interfície de *Java* y el *runtime engine*, las clases que pertenecen al proyecto ***compss-api*** son de la interfície *Java*, hemos marcado con verde las clases nuevas, esto servirá para que el desarrollador que continúe el trabajo conozca el estado actual del desarrollo.

Proyecto	Clase
compss-api	Binary.java
	BinaryContainer.java
	Container.java
	Containers.java
compss-agent	Loader.java
compss-loader	LoadersUtils.java
compss-engine	ITFParse.java
	COMPSsRuntimeImplTest.java
compss-commons	BinaryImplementation.java
	ContainerImplementation.java
	MethodType.java
	BinaryDefinition.java
	ContainerDefinition.java
	ImplementationDefinition.java
	BinaryContainerDescription.java
	ContainerDescription.java
compss-resources-commons	CoreManager.java
compss-adaptor-execution	Executor.java
	BinaryInvoker.java
	BinaryRunner.java

Tabla 2.3: Resumen de clases nuevas (en verde) y modificadas (en naraja) en la interfície y el *runtime*.

Por último, con el código incorporado en el *runtime* y la interfície de *Java* funcionando correctamente, estábamos listos para enlazar *PyCOMPSs* con la interficie *Java* del *framework*. Antes de empezar con la integración de este bloque, hemos repetido el proceso de pruebas para poder familizarme con el lenguaje python y entender el uso de los métodos con la anotación ***@binary***. Una vez terminado este proceso de adaptación de mi persona al funcionamiento de *PyCOMPSs*, finalmente hemos realizado la integración en el binding de Python modificando la anotación ***@binary*** y creando la nueva anotación ***@container***.

En la Tabla 2.4 tenemos las clases creadas y modificadas en el proyecto ***compss-bindings-python***, hemos marcado nuevamente en verde las clases nuevas.

Proyecto	Clase
compss-bindings-python	binary.py
	container.py
	task.py
	information.py

Tabla 2.4: Resumen de clases nuevas (en verde) y modificadas (en naranja) en la interfície de *PyCOMPSs*

2.2.4. Flujo de la integración en el *runtime*

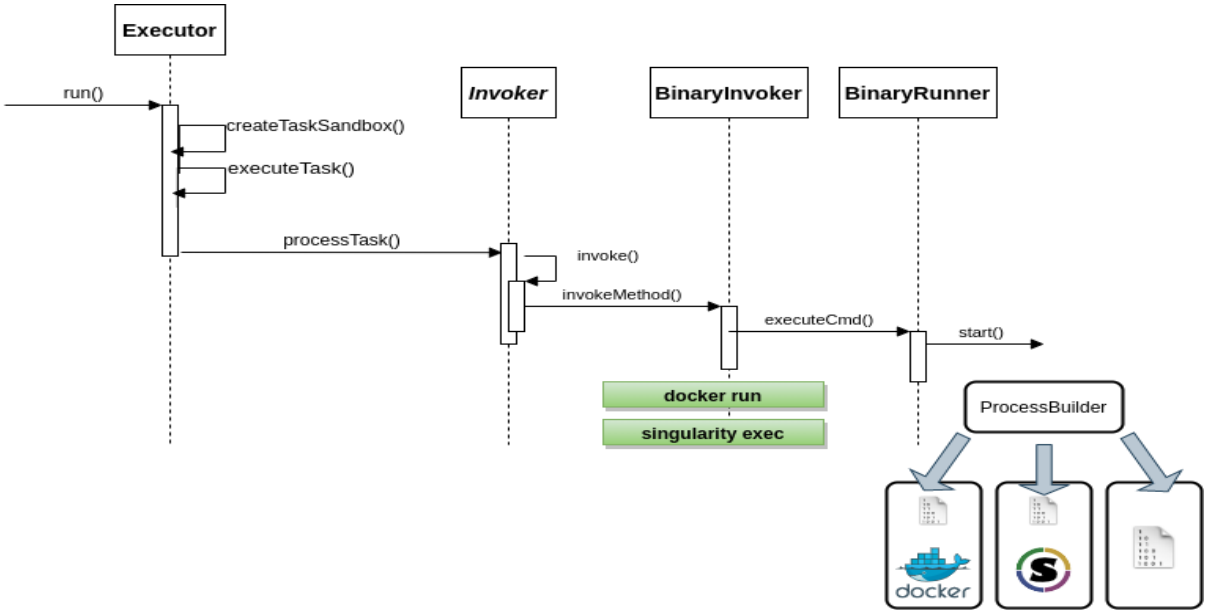


Figura 2.10: Diagrama de secuencia para la ejecución de tareas binarias en un Worker

A continuación detallaremos el diagrama de flujo (Figura 2.10) para lanzar tareas binarias en los Workers y los cambios que hemos realizado dentro del proyecto ***compss-adaptor-execution***.

Cuando el *runtime* que se ejecuta dentro de un Worker recibe una tarea para su ejecución, se instancia la clase *Executor* para procesar la tarea recibida; éste crea un *sandbox*[1] para el tipo de tarea que se ejecutará y lo envía a la función ***executeTask()***, que es donde se prepara la invocación de la tarea con la implementación del método, el directorio de trabajo y los recursos asignados. Finalmente se procesa la tarea para su ejecución instanciando la clase *BinaryInvoker*. Los cambios que hemos realizado en el *Executor* se hicieron para la creación del *sandbox* y la ejecución de la tarea (***createTaskSandbox()*** y ***executeTask()***), donde simplemente tuvimos que añadir la implementación para invocar el método de tipo *Container*, pasando los parámetros de la anotación, el directorio de ejecución y los recursos asignados para las tareas contenerizadas.

El *BinaryInvoker* simplemente ejecuta el método abstracto ***invoke()*** para procesar la implementación del método que hemos recibido desde el Executor. En el método ***invokeMethod()*** es donde preparamos el comando donde se ejecutará el binario y calculamos donde irán redirigidos los streams de entrada y salida de la tarea. Aquí se ha implantado la llamada a los comandos ***docker run*** y ***singularity exec*** que hemos estudiado previamente. Hay que tener en cuenta que *Docker* y *Singularity* en la línea de comandos también son binarios y se ejecutan seguido de sus parámetros; que en este caso serán sus opciones propias y el binario que se quiere ejecutar en la tarea. Entonces dependiendo del parámetro *engine* seleccionado por el programador, montaremos el comando para ***DOCKER*** o ***SINGULARITY***; en caso que este parámetro llegue ***[unassigned]***, no se utilizará ningún contenedor para su ejecución respetando la implementación anterior. Se integrarán en estos comandos las opciones que hemos seleccionado anteriormente para la ejecución de las tareas.

[1] Es un directorio de ejecución o espacio de trabajo que se utiliza para leer y escribir datos que se utilizan en la tarea.

Entre las opciones más importantes de los comandos ***docker run*** y ***singularity exec*** que hemos implementado, tenemos los bindings de montaje; estos nos sirven para acceder directamente al espacio de trabajo durante la ejecución de las tareas. Con esta opción podemos montar el directorio de trabajo del Worker de COMSPSs en el contenedor para poder acceder al espacio directamente, evitando leer y escribir datos en ubicaciones intermedias dentro de los contenedores. Otra opción es ***-i*** de *Docker* para redireccionar la entrada estándar *STDIN*. En la Tabla 2.2 podemos ver las opciones de los comandos que integraremos en la clase *BinaryInvoker*.

Una vez que la clase *BinaryInvoker* tiene el comando configurado junto con sus parámetros y se ha establecido el directorio donde se ejecutará y se accederá a los datos de la tarea, se instancia la clase *BinaryRunner*; ésta utiliza el método ***executeCmd()*** para lanzar la tarea binaria en el Worker. Para este propósito, en ***executeCmd()***, se utiliza la clase *ProcessBuilder* [27] que facilita el envío de comandos a través de la línea de comandos; maneja los recursos de entrada estándar, el destino de la salida estándar y la redirección de la salida estándar de error. Antes de lanzar la ejecución establecemos el entorno del proceso, usando el método ***environment()*** del *ProcessBuilder* para configurar las variables que necesitaremos en este entorno. Preparado el entorno para nuestro proceso, ejecutamos el método ***start()*** del *ProcessBuilder* para lanzar el comando e iniciar la tarea.

En el caso que el comando recibido sea ***docker run***, se ejecutará el binario dentro del contenedor de ***DOCKER*** utilizando las dependencias dentro de éste, si el comando que se ejecuta es ***singularity exec*** pasaría lo mismo, se ejecutaría en el contenedor de ***SINGULARITY***. Si el comando no viene contenerizado, el binario se ejecutará directamente en el sistema operativo del Worker utilizando sus dependencias, tal y como venía funcionando en versiones anteriores. Cuando las tareas finalizan, se activara el mecanismo de sincronización para transportar los datos de salida del espacio de trabajo hacia el Master.

Entre los requisitos para que COMPSs y la integración que hemos realizado funcione correctamente, debemos tener instalado *Docker* [15] y/o *Singularity* [17] en los Worker donde se ejecutarán las tareas binarias; si se utiliza el Master para este propósito, también debe tener instaladas ambas plataformas. La instalación de Docker debe estar configurada para ser ejecutada por todos los usuarios y evitar la autenticación [28]. En el caso de Singularity no hay ningún requerimiento especial para la instalación [29].

En la Figura 2.11 podemos ver como interactúan las diferentes partes en el flujo de ejecución para la contenerización de tareas binarias integrada dentro del *framework*.

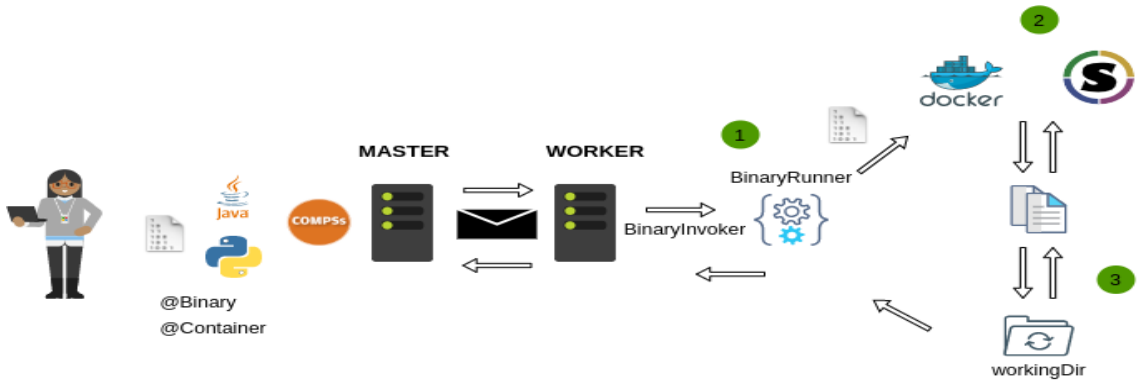


Figura 2.11: Flujo de la implemetación

3

Evaluación

En primer, lugar describiremos la aplicación que hemos utilizado para realizar la evaluación. A continuación detallaremos las evaluaciones que hemos realizado, para validar que nuestra integración funciona; estas serían: la comparativa de escalabilidad entre la aplicación contenerizada en *SINGULARITY* y sin contenerizar; un experimento para calcular el coste de despliegue que genera un contenedor y para finalizar, un análisis sobre la mejora en la productividad que la integración aporta a los programadores.

3.1. Aplicación con binarios Gromacs

La aplicación que utilizaremos se llama *«Lysozyme in Water»*, que establece un sistema de simulación que contiene una proteína (lisozima) en una caja de agua con iones. La aplicación se ha desarrollado en *PyCOMPSs* siguiendo un esquema de simulación [30] realizado por Justin A. Lemkul del departamento de Bioquímica de Virginia, USA.

La simulación consiste en generar para cada proteina una topología, posteriormente definir la caja y el solvato, agregar iones, procesar un mínimo de energía y genererar la gráfica de este mínimo obtenido. Para cada tarea de la aplicación hemos utilizado los binarios del paquete de simulación Gromacs [31], que sirve para realizar dinámicas moleculares, es decir, simular las ecuaciones de movimiento newtonianas para sistemas con millones de partículas. Para procesar cada proteína, se utilizan 9 tareas binarias hasta generar el gráfico y al finalizar la simulación, se ensamblan las imágenes resultantes con una única tarea en una imagen.

Para realizar la comparativa de escalabilidad se han escrito dos versiones de la aplicación; una versión con las tareas binarias contenerizadas, en la que utilizaremos dos imágenes de singularity para los binarios y otra versión de la aplicación sin contenerizar ninguna de las tareas. Las imágenes que utilizaremos son dos: una para utilizar los binarios de Gromacs y otra para ejecutar el binario de grace, que es una herramienta para gráficos que nos permitió pasar los ficheros de formato *xvg* a *png*. La imagen de Gromacs la hemos obtenido de su cuenta *DockerHub* [32] y hemos elegido la última versión de este paquete, la *2019.1*.

Gracias a que los formatos de ambas plataformas son compatibles, solo hemos tenido que crear los ficheros ***.def*** [33] utilizando imágenes de *DOCKER* y pasarlas a formato de imagen *SINGULARITY*. En la Figura 3.1 podemos ver los ficheros ***.def*** para la creación de las imágenes. En el lado izquierdo tenemos el fichero ***compss-gromacs.def*** y a la derecha el fichero ***compss-grace.def***.

Para finalizar, cabe destacar que se ha realizado la aplicación utilizando *PyCOMPSs* para validar que tanto la interfície de *Java*, como los bindings de *Python* funcionan correctamente después de la integración.

```
Bootstrap: docker
From: gromacs/gromacs
```

```
Bootstrap: docker
From: ubuntu

%post
    apt-get -y update
    apt-get -y upgrade
    apt-get -y install cmake gcc g++ grace

%environment
    export LC_ALL=C
```

Figura 3.1: Ficheros ***.def***, tenemos el fichero ***compss-gromacs.def*** (isquierda) y el fichero ***compss-grace.def*** (derecha)

3.2. Entorno para la evaluación

Para realizar la comparativa de escalabilidad se ha utilizado MareNostrum4 [34], que es un supercomputador basado en procesadores *Intel Xeon Platinum* de la generación *Skylake*. La parte de la máquina utilizada para realizar las pruebas cuenta con 16 nodos Lenovo SD350, con 96 GB de RAM y 48 cpu's por nodo; tiene un sistema de ficheros distribuido GPFS [35] al cual podemos acceder desde todos los nodos. Para contenerizar las tareas de la aplicación *«Lysozyme in Water»*, hemos copiado las imágenes creadas en el directorio distribuido GPFS, para que los nodos puedan acceder a éstas durante las evaluaciones. Estas imágenes son ***compss-gromacs.sif*** y ***compss-grace.sif***, que fueron generadas a partir de los ficheros mencionados en la Figura 3.1

3.2.1. Evaluación de Gromacs

La evaluación con Gromacs ha consistido en comparar los resultados de la aplicación contenerizada con ***SINGULARITY*** y la aplicación sin contenerizar; por una parte haciendo un test de escalabilidad fuerte (*strong scaling*), que consistió en ejecutar la aplicación con una entrada fija de 16 proteínas incrementando la cantidad de nodos proporcionalmente; y por otra, hacer un test de escalabilidad débil (*weak scaling*), donde se aumentó proporcionalmente el tamaño del problema (proteínas) y el número de elementos de procesamiento simultáneo (nodos). Cabe recalcar que durante esta evaluación se han tomado 10 muestras de cada ejecución.

Strong Scaling

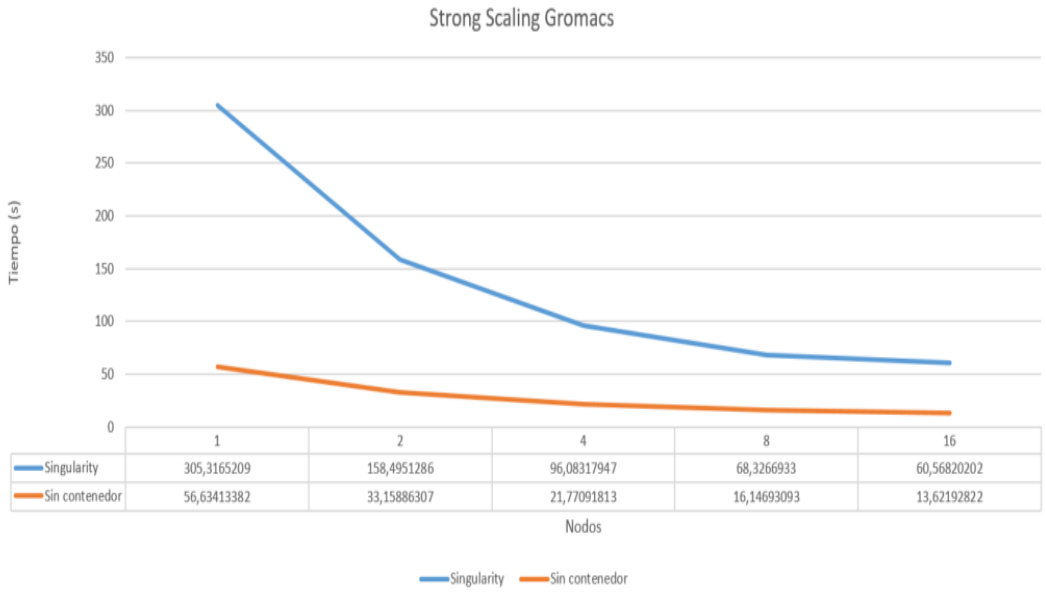


Figura 3.2: Tiempos de ejecución al aumentar el número de nodos

En la Figura 3.2 podemos ver que la gráfica tiene en el eje vertical el tiempo en segundos y en el horizontal el número de nodos. El tamaño del problema que hemos utilizado es de calcular 16 proteínas en paralelo. La línea azul indica la media de las muestras, para el mismo número de nodos con la aplicación contenerizada y la línea en color naranja indica la media de muestras para la aplicación no contenerizada. En los recuadros de abajo del gráfico, podemos ver la media en segundos del tiempo de ejecución para cada incremento de recursos.

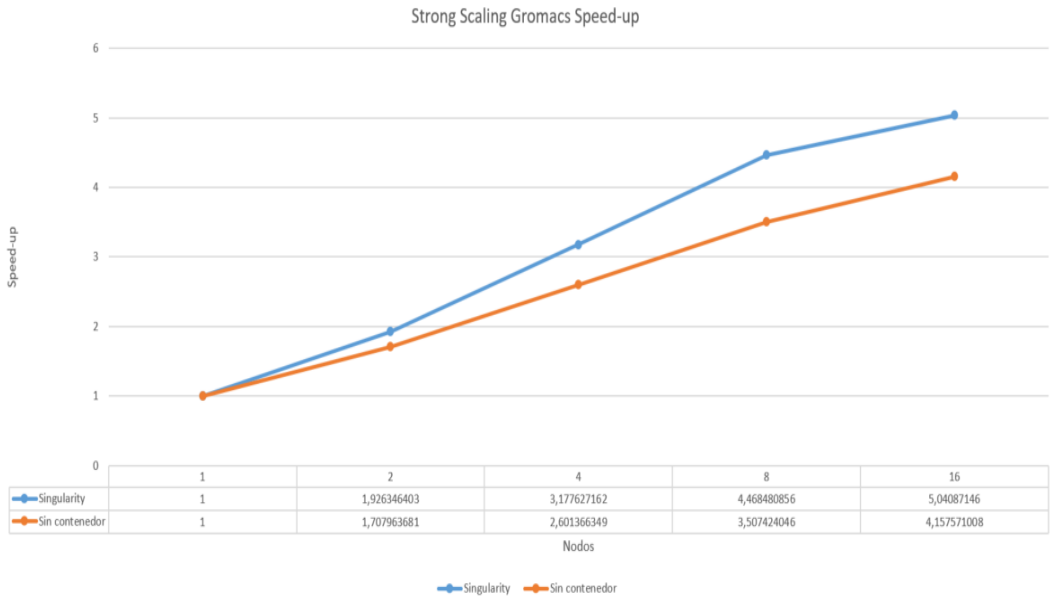
Para la aplicación contenerizada, vemos que esta no llega a escalar de forma ideal, pero tampoco lo esperamos ya que la parte que consume todos los recursos es el energy minimization, y una parte del merge final donde no existe paralelismo. De todas formas la escalabilidad no se degrada por usar contenedores, sino que la mejora; esto es debido a que al haber mas overhead, este se paraleliza al distribuirse la aplicación en más nodos.

Por otra parte, podemos apreciar que la diferencia es bastante notable entre ambas ejecuciones; la ejecución conteneriza tarda casi seis veces más, creemos que esto es debido a las optimizaciones hechas para la arquitectura e infrastructura de MareNostrum, cosa que en los contenedores no se ha hecho; como el compilador de intel e instrucciones vectoriales propias de la arquitectura, link a librerias de algebra lineal optimizadas o la lectura de la imagen en GPFS. Por falta de tiempo no se ha podido investigar más. En la tabla 3.1 podemos ver unas pruebas que se ejecutaron en local para medir la diferencia del tiempo de ejecución de las aplicaciones con 3 proteínas. Podemos ver que ejecutando la misma aplicación en la misma arquitectura los tiempos de ejecución no varían tanto.

DOCKER	SINGULARITY	Solo binario
73,170 seg.	57,312 seg.	50,336 seg.

Tabla 3.1: Media de la ejecución de «*Lysozyme in Water*» con 3 proteínas en local

En la gráfica de la Figura 3.3, la ganancia de ambas implementaciones nos indica que las aplicaciones no escalan de forma ideal, ya que al aumentar los recursos el tiempo de ejecución no se reduce proporcionalmente. Podemos ver que la solución con SINGULARITY se comporta bien hasta los 8 nodos, donde se aleja del ideal, como hemos dicho antes, atribuimos esto a la aplicación y a las optimizaciones hechas para la arquitectura.



	1	2	4	8	16
Singularity	1	1,926346403	3,177627162	4,468480856	5,04087146
Sin contenedor	1	1,707963681	2,601366349	3,507424046	4,157571008

Figura 3.3: Ganancia al aumentar el número de nodos

Weak Scaling

En un prueba de escalabilidad débil esperamos que la ganancia sea siempre igual, ya que el tamaño del problema irá creciendo proporcionalmente con los recursos, pero esto nunca se da debido al overhead del sistema. El ideal para esta prueba es que siempre sea constante en 1, pero podemos ver que mientras la solución no contenerizada se degrada totalmente, la solución contenerizada se estabiliza cuando tenemos 16 nodos.

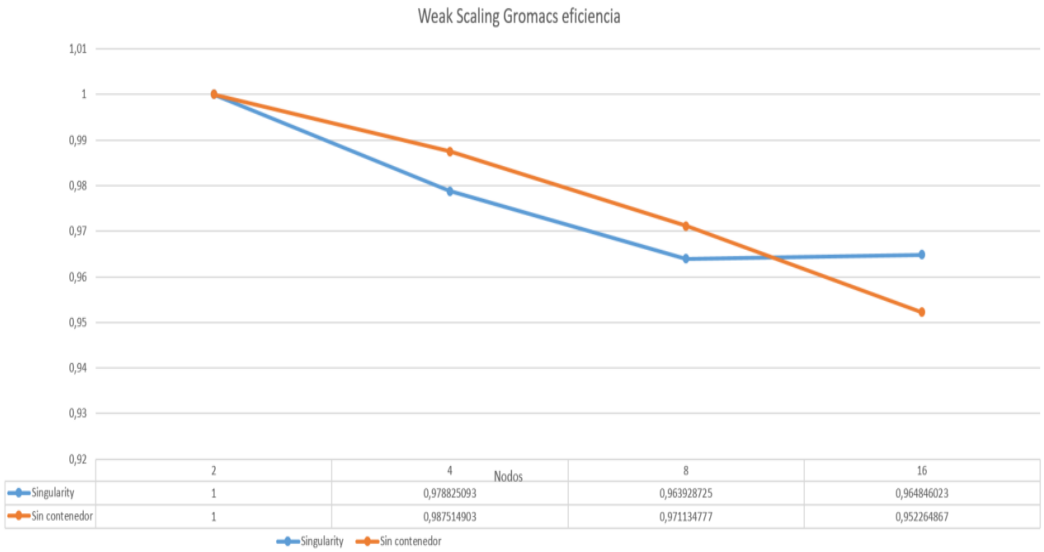


Figura 3.4: Eficiencia al mantener proporcional el tamaño del problema con el número de nodos

3.2.2. Coste del despliegue

La ejecución de una tarea no es instantánea, existe un coste de despliegue que es el tiempo que se tarda en inicializar y liberar todos los recursos necesarios, antes y después de ejecutar cualquier tarea o aplicación. El entorno de ejecución o el kernel necesitan reservar los recursos y liberarlos al terminar (por ejemplo, el sistema de colas del MareNostrum4). Este coste de despliegue puede resultar insignificante cuando se ejecuta el programa directamente sobre el anfitrión, realizando una ejecución nativa, pero es un hecho que el uso de contenedores lo aumenta considerablemente. Al usar contenedores para ejecutar la aplicación se tiene que virtualizar el entorno (crear el contenedor y destruirlo al terminar), por lo que el tiempo de ejecución total aumenta. Por esta razón, es necesario evaluar el coste de despliegue de los contenedores respecto a ejecuciones sin contenerizar.

Para evaluar el coste del despliegue, hemos hecho unas pruebas en local con una aplicación donde levantamos tres tareas diferentes; con DOCKER, SINGULARITY y sin contenerizar. No hemos podido realizar estas prueba en MareNostrum4 ya que los nodos no disponen de una instalación de DOCKER.

Dentro de cada tarea lanzamos el binario ***sleep*** configurado con 10 segundos, colocamos un *compss_barrier()* al final de cada tarea para capturar el tiempo y calcular la diferencia. En la Figura 3.5 presentamos el código con el que realizamos esta prueba. Se han tomado 10 muestras de cada ejecución durante esta evaluación.

```python
def main_program():
    from pycompss.api.api import compss_open

    # Check and get parameters
    if len(sys.argv) != 2:
        usage()
        exit(-1)
    initialValue = sys.argv[1]

    # Start counting time
    start_time = time()
    #Execute DOCKER_FUNC
    docker_func()
    compss_barrier()
    elapsed_time = time() - start_time
    print("Elapsed time from Docker: %0.10f seconds." % elapsed_time)


    # Start counting time
    start_time = time()
    # Execute SINGULARITY_FUNC
    singularity_func()
    compss_barrier()
    elapsed_time = time() - start_time
    print("Elapsed time from SINGULARITY: %0.10f seconds." % elapsed_time)


    # Start counting time
    start_time = time()
    # Execute BINARY_FUNC
    binary_func()
    compss_barrier()
    elapsed_time = time() - start_time
    print("Elapsed time from ONLY BINARY: %0.10f seconds." % elapsed_time)



@container(engine="DOCKER", image="ubuntu", binary="sleep")
@task()
def docker_func(time='10'):
    pass

@container(engine="SINGULARITY",
           image="/home/compss/singularity/examples/ubuntu_latest.sif",
               binary="sleep")
@task()
def singularity_func(time='10'):
    pass

@binary(binary="sleep")
@task()
def binary_func(time='10'):
    pass
```

Figura 3.5: Aplicación utilizada para calcular el coste de despliegue de una tarea

DOCKER	SINGULARITY	Solo binario
2,340 seg.	0,396 seg.	0,137 seg.

Tabla 3.2: Tiempo en segundos del coste de despliegue de los contenedores DOCKER y SINGULARITY

En la tabla 3.2 tenemos los resultados de la evaluación del coste de despliegue, a simple vista podemos comprobar que el uso de contenedores afecta negativamente al coste del despliegue de la tarea. Esta claro que el coste de levantar SINGULARITY es muy pequeño. En cambio el coste de levantar DOCKER es más elevado cosa que podríamos atribuirlo a su arquitectura y complejidad. Éste utiliza un proceso *daemon* para funcionar que requiere escalar permisos para iniciar su despliegue.

3.2.3. Coste en productividad

Vamos a intentar mostrar el impacto que tiene nuestra solución en la productividad de un programador, comparando las líneas de código que tenía que preparar antes de la integración y las que debe realizar ahora para contenerizar un binario. Antes de la integración el programador no podía contenerizar un binario directamente con la etiqueta @Binary, debía hacerlo a través de una tarea nativa.

Por ejemplo, imaginad que queremos contenerizar el binario ***ls***, imprimir el resultado por la salida estándaro o el estandar error. En la Figura 3.6 podemos ver que era necesario escribir 3 lineas de código para lanzar la instrucción docker run, y 15 más solo para la salida estándar y error por consola. En la figura 3.7 vemos que con la etiqueta ***@Container*** solo necesitamos pasar el nombre de la plataforma, la imagen y el binario. La salida estandar y error se generan automáticamente en la salida en la consola sin necesidad de escribir una sola línea de código. En conclusión, nos hemos ahorrado 18 líneas de código con la etiqueta @Container lo cual nos da una visión clara de la mejora que aportamos al programador con esta integración.

```
public static void dockerDirectCommandExecute() throws FileNotFoundException, IOException, InterruptedException {
   File stderrFile = File.createTempFile("tmpErr", "out");
   File stdoutFile = File.createTempFile("tmpStd", "out");
   try {
      ProcessBuilder builder = new ProcessBuilder("docker run --rm ubuntu ls");
      Process p = builder.start();
      int exitCode = -1;
      boolean done = false;
      while (!done) {
         try {
            exitCode = p.waitFor();
            done = true;
         } catch (InterruptedException ie) {
            System.out.println("Interrupted waiting for process to exit.");
         }
      }
      BufferedReader err = new BufferedReader(new FileReader(stderrFile));
      BufferedReader in = new BufferedReader(new FileReader(stdoutFile));
      ................
   } finally {
      stderrFile.delete();
      stdoutFile.delete();
   }
}
```

Figura 3.6: Líneas de código para contenerizar un binario desde un método nativo

```
@Container(engine="SINGULARITY", image="/home/compss/singularity/examples/ubuntu_latest.sif",
                                                    binary="ls", workingDir="/test")
void pwdContainer(
        @Parameter(type = Type.FILE, direction = Direction.OUT, stream = StdIOStream.STDOUT) String out
        );
```

Figura 3.7: Contenerizar un binario con la etiqueta @container

4

Conclusiones y trabajo futuro

Con este trabajo he aprendido a utilizar el modelo de programación de COMPSs para desarrollar aplicaciones y he visto desde una perspectiva diferente, las necesidades que tiene un desarrollador al trabajar en plataformas distribuidas para aumentar su productividad. Con la integración hemos hecho una adaptación a las anotaciones binarias existentes y proporcionado nuevas a la interfície de *Java* y *PyCOMPSs*, que facilitarán el desarrollo de software contenerizado en los flujos de tareas binarias.

Con respecto a los componentes internos de COMPSs, hemos realizado modificaciones en la interfície y en los ejecutores de tareas binarias, estudiando previamente el flujo de éstas para no romper el funcionamiento de versiones anteriores. Las modificaciones realizadas permiten levantar contenedores, utilizando los comandos de las dos plataformas escogidas, preservando la ejecución de binarios nativa anterior. He aprendido el funcionamiento y diferencias de las plataformas integradas en COMPSs así como utilizar las compatibilidades entre ambas.

Para evaluar nuestra integración, hemos implementado una aplicación basada en un workflow de Gromacs y la hemos ejecutado en local utilizando el uso de binarios y con contenedores de DOCKER y SINGULARITY. Hemos visto que el overhead de DOCKER es mayor que el de SINGULARITY. También hemos evaluado el rendimiento y la escalabilidad de la aplicación en MN comparando la ejecución de binarios directa con la de contenedores de SINGULARITY. En este caso hemos visto que aunque el tiempo de ejecución es mucho mayor en containers, este no afecta la escalabilidad de la aplicación, ya que el overhead producido por los contenedores se distribuye con las tareas. Por otra parte, se puede concluir se ha mejorado la productividad de los programadores, reduciendo el código que estos deben generar para contenerizar tareas binarias en sus desarrollos utilizando una sola línea.

En lo personal, este trabajo me ha dado la oportunidad de poder realizar un desarrollo dentro de un framework como COMPSs y ver las cosas desde otra perpectiva. He tenido la oportunidad de utilizar las máquinas que me ha asignado el BSC en el MareNostrum4 y materializar los conociemientos adquiridos durante la carrera sobre sistemas distribuidos y paralelismo.

Como trabajo futuro, sería fantástico integrar en los bindings de C/C++ las anotaciones que hemos implementado, ya que por falta de tiempo no hemos podido finalizar; estas modificaciones serían muy similares a las que ya hemos realizado con Java y Python en la interfície de usuario. Estudiar la manera de reducir el coste de despliegue en las tareas que genera esta integración. Por último, integrar otras plataformas de contenedores como *LXC* [36] o *rkt* [37] y crear un ejecutor independiente para las tareas ***@container***.

5

Informe de sostenibilidad

5.1. Dimensión económica

1. **¿Has cuantificado el coste (recursos humanos y materiales) de la realización del proyecto? ¿Qué decisiones has tomado para reducir el coste? ¿ Has cuantificado este ahorro? Si hicieras de nuevo el proyecto, ¿Podrías realizarlo con menos recursos?**

   No ha dado tiempo de hacer esta cuantificación, pero considerando el coste calculado en GEP, y que no hemos tenido ninguna desviación, podemos decir que los costes han sido prácticamente los definidos allí.

   Para reducir el coste hemos intentado utilizar una porción de las máquinas disponibles asignadas por el BSC y minimizar el uso de éstas. Se intentado evitar los gastos de transporte realizando las reuniones vía telemática.

   Dudo que sea posible realizar el proyecto con menos recursos de los utilizados actualmente, ya que son los imprescindibles.

2. **¿Qué costes estimas que tendrá el proyecto durante su vida útil? ¿Se podría reducir este coste para hacerlo más viable**

   El proyecto necesita estar en un repositorio para su mantenimiento, por tanto, tendría costes de alojamiento en la nube (GitLab) durante su vida útil. También tiene gastos en recursos humanos y gastos de oficina donde los desarrolladores realizan el mantenimiento. No se ha podido cuantificar los costes que puede generar mantener este *framework*.

   Una manera de reducir los costes es generando ingresos realizando consultoría a empresas que utilicen el *framework*.

3. **¿Podrían producirse escenarios que perjudicasen la viabilidad del proyecto?**

   El proyecto de COMSPSs es de código abierto por lo que la viabilidad no podría verse afectada. La integración realizada en este proyecto pretende ser utilizada en investigaciones científicas y estudios con subvención nacional y europea.

5.2. Dimensión social

1. **¿La realización de este proyecto ha implicado reflexiones significativas a nivel personal, profesional o ético de las personas que han intervenido?**

   Las reflexiones realizadas han sido sólo a nivel profesional y relacionadas al proyecto.

2. **¿Quien se beneficiará del uso del proyecto? ¿En qué medida soluciona el proyecto el problema planteado inicialmente?**

   Todos los desarrolladores que lo utilicen, principalmente está destinado a personal científico y académido de ingeniería.

   El proyecto aporta productividad a los programadores que utilicen el *framework* para desarrollar aplicaciones donde esten implicados binarios de diferentes librerías.

5.3. Dimensión ambiental

1. **Si hiciera de nuevo el proyecto ¿Podrías realizarlo con menos recursos?**

   Sí, conociendo la metodología con que se realizan las integraciones, sería capaz de realizar el proyecto con los recursos imprescindibles.

2. **¿Qué recursos estimas que tendrá el proyecto durante su vida útil? ¿Cuál será el impacto ambiental de estos recursos?**

   Al ser un *framework*, el proyecto siempre estará en el contexto de otro proyecto. Así que la vida útil del *framework* será inherente al proyecto o usuarios que lo utilicen. Por otra parte, mientras el *framework* este en desarollo necesitará recursos en repositorios y servidores, por lo que el impacto ambiental vendrá derivado del consumo de estos servidores.

Apéndice

A

Planificación Temporal

El TFG tuvo una duración aproximada de 4 meses y 2 semanas, desde mediados de febrero hasta finales de junio (del 29 al 3 de julio). Se estimaron unas 475 horas aproximadas para desarrollar, documentar, hacer la memoria y preparar la defensa. Estas 475 horas se han distribuido entre las siguientes tareas que mencionaremos a continuación.

A.1. Descripción de las tareas

A.1.1. Planificación del proyecto

Corresponde a la realización de la documentación de gestión del proyecto (GEP) que empezó el 17 de febrero de 2020 y finalizó el día 16 de marzo. Fue una de las partes más importantes del proyecto ya que se definió todos los detalles del TFG. Esta fase no tuvo ninguna dependencia, pero en cambio, tuvo unas fechas de entrega que se debían cumplir obligatoriamente.

A.1.2. Fase Inicial

Esta fase tuvo una duración de cuatro semanas con una previsión de trabajo de unas 18 horas por semana; entre el día 17 de febrero al día 16 de marzo. Se compone de la siguiente tareas:

1. **Análisis de requisitos:** Se definió con el codirector, los detalles de como debía ser la interacción del usuario/programador con la nueva funcionalidad que se queria implantar. Se calcula que esta tarea llevó unas 15 horas.

2. **Diseño del prototipo:** Una vez finalizado el análisis, se paso a estudiar el diseño técnico y la arquitectura de COMPSs, para tener claro los flujos y no romper el código durante el desarrollo. Una vez se tuvo clara la arquitectura se realizó el diseño del prototipo que integramos. Se estimaron 30 horas para realizar esta tarea.

3. **Preparación del entorno de desarrollo:** Instalar en local todo el software y herramientas necesarias para empezar a programar, realizar pruebas y experimentar con el framework. Para esta tarea se estimaron un total de 30 horas.

A.1.3. Fase de Desarrollo

Para la fase de desarrollo se planificaron 3 iteraciones y se estimaron 25 horas de dedicación semanal.

Primera iteración

La primera iteración ha consistido en *definir e implementar las anotaciones nuevas, para la ejecución de binarios en contenedores y extender las anotaciones ya existentes en la interfície de COMPSs.*

Segunda iteración

En la segunda iteración se investigó *el funcionamiento y flujo de las ejecuciones de binarios, para crear y modificar las clases e interfaces necesarias para el funcionamiento de la nueva integración.*

Tercera iteración

En la tercera iteración, se investigó el funcionamiento de Docker y Singularity. También se instalaron ambas plataformas de contenedores en local y se experimentó con ellas para integrarlas dentro de COMPSs.

A.1.4. Fase de validación final

En esta fase, se validó en local el funcionamiento de la nueva integración, y se realizó una presentación con la directora del proyecto para aprobar la fase de integración. Luego, se procedió a realizar pruebas de la integración con una aplicación real. Por último, lanzamos la nueva aplicación en el entorno de pruebas de COMPSs en Marenostrum4 [34] para realizar las evaluaciones.

A.1.5. Fase Final

Una vez finalizadas las tareas de desarrollo y validación, se redactó la memoria y se preparó la defensa del trabajo. Se estimó 60 horas para esta fase.

A.2. Tiempo estimado

En la tabla A.1 proporcionamos un resumen con las horas que se estimaron para cada fase.

Fase	**Tiempo total (horas)**
Planificación del proyecto	75
Fase inicial	70
Fase de desarrollo	180
Fase de validación final	60
Fase final	90
Total	**475**

Tabla A.1: Horas estimadas del proyecto

A.2.1. Diagrama de la planificación del proyecto

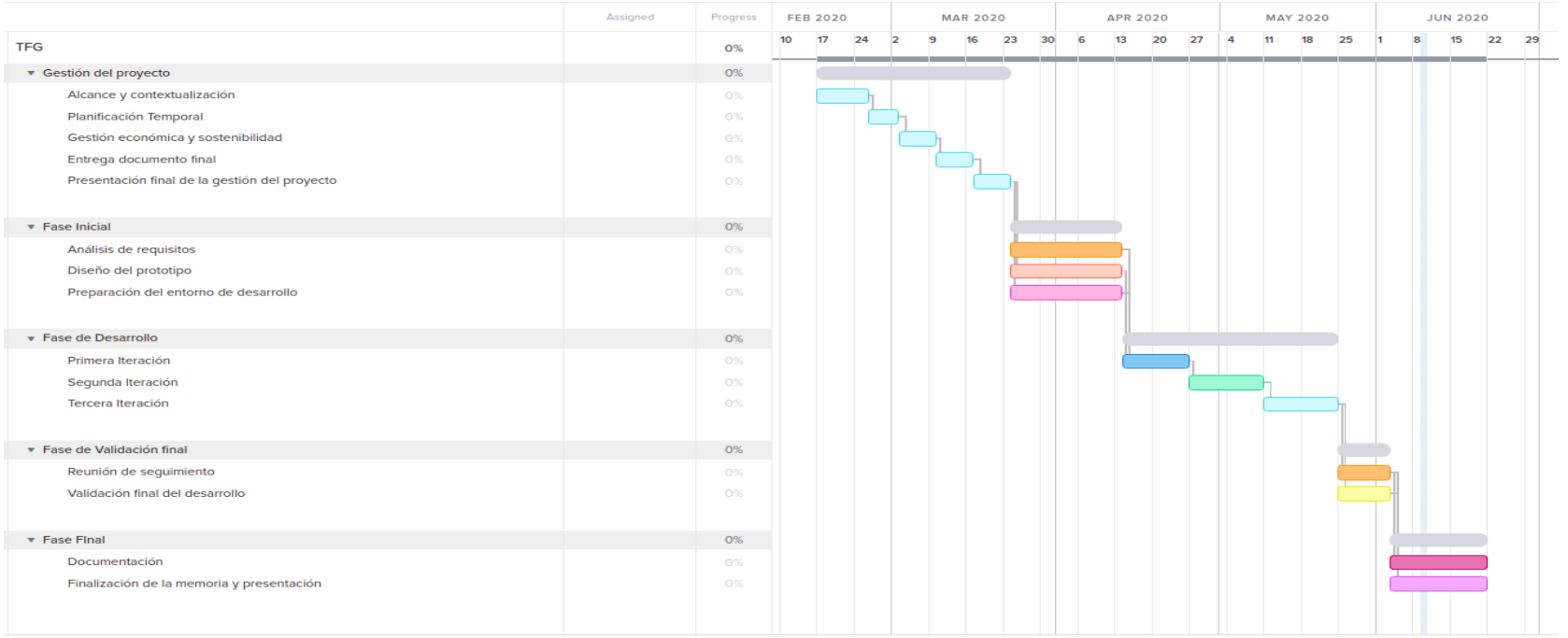


Figura A.1: Diagrama de Gantt

A.3. Recursos

A continuación se detallan los recursos que fueron necesarios para el correcto desarrollo del proyecto:

A.3.1. Recursos humanos

- **Autor del proyecto:** Asumió todos los roles necesarios para el desarrollo y dedicó una media de 20 horas semanales.
- **Directores del proyecto:** Han colaborado con el autor del proyecto validando las tareas y corrigiendo la documentación del TFG.

A.3.2. Recursos hardware

- **Ordenador portátil:** Acer TravelMate P259 Intel Core I5-6200U, 12GB de RAM, Ubuntu 18.04.4 LTS.

A.3.3. Recursos software

A.4. Valoración de alternativas y plan de acción

Durante el proceso del proyecto, tuvimos desviaciones desviaciones por bloqueos durante el desarrollo que retrasaron las tareas planificadas. Estas fueron:

A.4.1. Estimaciones incorrectas

Al inicio del proyecto teníamos estimado utilizar dos aplicaciones reales para realizar las evaluaciones, pero debido a estimaciones incorrectas en la tercera iteración, decidimos realizar las pruebas con una sola aplicación, para poder terminar satisfactoriamente el proyecto en el tiempo inicialmente estimado.

B

Gestión económica

Los costos del proyecto están directamente relacionados con los recursos de hardware, software y humanos, todos los descritos en la planificación temporal. A continuación se hará un estudio del coste del proyecto que, a pesar de no tratarse de un proyecto con inversión real, se estimó como si lo fuese.

B.1. Recursos humanos

El proyecto fue desarrollado por una persona que desempeñó todos los roles que pueden participar en él. Para el cálculo de la relación salarial de horas por rol desempeñado, nos basaremos en el estudio de remuneración de PagePersonnel del 2018 [38], considerando la media de los salarios brutos de como máximo 3 años de experiencia. A continuación, en la tabla B.1 podemos ver el costo por horas de los diferentes roles.

Rol	Precio por Hora
Project Manager	40€
Analista	35€
Arquitecto	35€
Programador	20€
Tester	20€

Tabla B.1: Costo por horas de los diferente roles

A continuación se detalla el cálculo del coste en recursos humanos para la actividad planificada en el diagrama de Gantt, mostrando para cada una de las tareas, el rol que se ha realizado y el número de horas que se dedicó. Cada uno de estos cálculos se ha realizado basándose en los costes de la Tabla B.1, en la Tabla B.2 podemos ver el desglose del coste:

Fase	Tarea	Horas	Recurso	Coste
Gestión del proyecto	Alcance y contextualización	20	Project Manager	3000€
	Planificación temporal	20		
	Gestión económica y sostenibilidad	20		
	Presentación de documento final	10		
	Presentación final	5		
Fase inicial	Análisis de requisitos	20	Analista[50 %] Arquitecto[50 %]	700€
	Diseño del prototipo	20	Analista[50 %] Arquitecto[50 %]	700€
	Preparación del entorno de desarrollo	30	Programador	600€
Fase de Desarrollo	Iteraciones	60	Programador	1200€
		60	Programador	1200€
		60	Programador	1200€
Validación final	Validación final del desarrollo	60	Tester	1200€
Fase Final	Documentación	30	Project Manager	3600€
	Finalización de la memoria y presentación	60		
Total		475		13400€

Tabla B.2: Costes directos por actividad

B.2. Recursos hardware y software

En este proyecto solo se utilizó un portátil Acer TravelMate P259 Intel Core I5-6200U de 12GB de RAM con Ubuntu 18.04. Cabe mencionar que la amortización es el valor a calcular teniendo en cuenta que el proyecto se desarrolló en 4,2 meses. El coste de amortización fue de 74,54€.

Los recursos software mencionados anteriormente en el plan temporal fueron totalmente gratuitos. En la Tabla B.3 podemos ver la estimación de costes de recursos software y hardware :

Recurso	Precio	Unidades	Vida Útil	Amortización
Acer TravelMate P259	841,89€	1	4	74,54€
GitLab	0€	1	-	0€
Eclipse	0€	1	-	0€
VisualStudio Code	0€	1	-	0€
Latex	0€	1	-	0€
Google Documents	0€	1	-	0€
TeamGantt	0€	1	-	0€
Total	841,89€	8	-	74,54€

Tabla B.3: Estimación de costes de recursos software y hardware

B.3. Costes indirectos

Se tuvo en cuenta otros gastos como:

- Consumo energético del ordenador portátil.

- Servicio de conexión a Internet con 50 MB de bajada y 50 MB de subida.

Descripción	Precio	Cantidad	Costo estimado
Consumo energético	0,15€/kWh	Consumo diario de 1,5 kWh por 135 días	30,37€
Internet	50€/mes	4 meses y medio	225€
Total			255,37€

Tabla B.4: Estimación de costes indirectos

B.4. Contingencias e imprevistos

Para poder hacer frente a posibles imprevistos que podían surgir durante de la realización del proyecto, se destinó un margen de la suma del 15 % de costes directos e indirectos. En la Tabla B.5 podemos ver un resumen de la partida de costes e imprevistos.

Concepto	Coste	Porcentaje de contingencia	Coste de contingencia
Costes directos	13400€	15 %	2010€
Costes indirectos	255,37€	15 %	38,30€
Fallo ordenador	100€	10 %	10€
Problemas con servicios externos	68€	5 %	3,4€
Desviación por mal diseño	1380€	40 %	552€
Total			2613,70€

Tabla B.5: Partida de contingencia e imprevistos

Los potenciales imprevistos detectados en el proyecto fueron:

- *Avería del ordenador principal donde se desarrollaba el proyecto.* Aunque la probabilidad de avería fue muy baja, se tuvo que tener en cuenta los posibles fallos de hardware y afrontar su reparación o substitución, así como también el tiempo de instalación de todo el entorno de desarrollo. El costo previsto de reparación fue de 100€con un 10 % de probabilidad que suceda.

- *Posible pérdida de datos.* La probabilidad fue remotamente baja, ya que se utilizaron servicios externos para almacenar el código y los documentos (GitLab y Google Drive), así que no consideramos realmente este imprevisto.

- *Problemas con servicios externos (conexión eléctrica e Internet)*, lo que podía provocar no poder desarrollar el proyecto durante las horas de avería. La probabilidad estimada fue del 5 % y en el caso de que esto ocurriera se asumió que se perdería un día de trabajo. Por lo tanto, con una estimación de 4-6 horas de trabajo diario y con una media de coste de los diferentes roles de 17€/ hora, la estimación del imprevisto fue de 68€.

- *Desviación por mal diseño.* El peor de los casos que hemos estimado anteriormente, es que se haya realizado un mal diseño de la arquitectura y detectarla estando muy adelantados en las tareas de implementación. Esto nos habría obligado a utilizar las 3 semanas destinadas a terminar el proyecto.

Estas 90 horas las distribuiríamos en 60 como rol de desarrollador y 30 como arquitecto, que sería un presupuesto de 1380€. Creemos que existió una probabilidad media del 40 % que esto ocurriera debido a la inexperiencia que tenemos.

B.5. Presupuesto total

En la Tabla B.6, se muestra un resumen de todos los costes mencionados anteriormente y el coste total de la realización del proyecto.

Recursos	Coste estimado
Recursos humanos	13400€
Recursos software y hardware	74,54€
Costes indirectos	255,37€
Contingencias e imprevistos	2613,70€
Coste total	16343,61€

Tabla B.6: Costo total del proyecto

B.6. Control de gestión

Fue necesario llevar un control de los recursos para ver si era necesario el uso del plan de contingencia. Para el caso de los recursos humanos, se llevó un registro de las horas que suponían las tareas. Con este método se comprobó que se estaba cumpliendo con la planificación. Para el caso del recurso hardware, el gasto incluido anteriormente, fue fijo, ya que es el precio de venta al público establecido por las marcas de los productos, por tanto, en este aspecto no tuvimos ninguna sorpresa. En cuanto al coste total del producto, no hubo ningún cambio respecto a la planificación inicial. Para el caso de los recursos software, no tuvimos ningún gasto incluido, por lo tanto, en este aspecto tampoco tuvimos ninguna sorpresa.

Bibliografía

[1] Workflows and Distributed Computing Group. URL: https://www.bsc.es/discover-bsc/organisation/scientific-structure/workflows-and-distributed-computing.

[2] Barcelona Supercomputing Center (BSC). URL: https://www.bsc.es/.

[3] COMP Superscalar, COMPSs. URL: https://compss-doc.readthedocs.io/en/2.6/Sections/00_Intro.html.

[4] Docker Swarm,Docker. URL: https://docs.docker.com/engine/swarm/.

[5] PyCOMPSs 2.6. URL: https://pypi.org/project/pycompss/.

[6] Barcelona Supercomputing Center (BSC). Extrae Tool. URL: https://tools.bsc.es/extrae.

[7] Barcelona Supercomputing Center (BSC). Paraver Tool. URL: https://tools.bsc.es/paraver.

[8] COMPs Superscalar Integrated Development Environment. URL: https://marketplace.eclipse.org/content/comp-superscalar-integrated-development-environment.

[9] Javassist,MVNrepository Singularity. URL: https://mvnrepository.com/artifact/javassist/javassist.

[10] A Practical Introduction to Container Terminology. URL: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introductio.

[11] Open Container Initiative. URL: https://opencontainers.org/faq/#n22.

[12] RunC. URL: https://github.com/opencontainers/runc.

[13] Container Image Format Specification. URL: https://github.com/opencontainers/image-spec/blob/master/README.md.

[14] Container Runtime Specification. URL: https://github.com/opencontainers/runtime-spec/blob/master/README.md.

[15] Docker Engine. URL: https://docs.docker.com/storage/bind-mounts/.

[16] Docker Engine. URL: https://docs.docker.com/get-started/overview/#docker-engine.

[17] Singularity Engine. URL: https://sylabs.io/guides/3.5/user-guide/bind_paths_and_mounts.html.

[18] MPI Standard Draft. URL: https://www.mpi-forum.org/.

[19] Trello. URL: https://trello.com/es.

[20] Use of Clean up (–rm), Docker. URL: https://docs.docker.com/engine/reference/run/#clean-up---rm.

[21] Use bind mounts, Docker. URL: https://docs.docker.com/storage/bind-mounts/.

[22] Use of WORKDIR, Docker. URL: https://docs.docker.com/engine/reference/run/#workdir.

[23] Foreground -i STDIN, Docker. URL: https://docs.docker.com/engine/reference/run/#foreground.

[24] Bind Paths and Mounts, Singularity. URL: https://sylabs.io/guides/3.5/user-guide/bind_paths_and_mounts.html.

[25] Parameters summary for Python Bindings, COMPSs. URL: https://compss-doc.readthedocs.io/en/2.6/Sections/02_User_Manual_App_Development.html#task-selection.

[26] Java annotated interface, COMPSs. URL: https://compss-doc.readthedocs.io/en/2.6/Sections/02_User_Manual_App_Development.html#java-annotated-interface.

[27] Class ProcessBuilder Java 8. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html.

[28] Install using the convenience script, Docker. URL: https://docs.docker.com/engine/install/ubuntu/#install-using-the-convenience-script.

[29] Install Singularity, Singularity. URL: https://sylabs.io/guides/3.5/user-guide/quick_start.html#quick-installation-steps.

[30] Lysozyme in Water, Justin A. Lemkul, Ph.D URL: http://www.mdtutorials.com/gmx/lysozyme/index.html.

[31] Gromacs. URL: http://www.gromacs.org/About_Gromacs.

[32] GROMACS molecular dynamics simulations, dockerhub URL: https://hub.docker.com/r/gromacs/gromacs.

[33] Definitions files , Singularity URL: https://sylabs.io/guides/3.5/user-guide/build_a_container.html?highlight=def#build-a-container.

[34] MareNostrum. URL: https://www.bsc.es/marenostrum/marenostrum.

[35] IBM General Parallel File System (GPFS) URL: https://www.ibm.com/support/knowledgecenter/en/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.biginsights.product.doc/doc/bi_gpfs_overview.html.

[36] LXC, linux containers URL: https://linuxcontainers.org/.

[37] rkt containers URL: https://coreos.com/rkt/docs/latest/.

[38] Pagepersonnel.es. (2019). Recuperado 2 octubre de 2019. URL: https://www.pagepersonnel.es/sites/pagepersonnel.es/files/PG_ER_IT_2018.pdf.